1

DEVELOPMENT OF THE KALMAN FILTER APPLICATION AND A VHDL MODEL FOR THE AFIT FLOATING POINT APPLICATION SPECIFIC PROCESSOR (FPASP)

THESIS

William E Koch
Capt, USAF

AFIT/GCE/ENG/89D-3

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCE/ENG/89D-3

DEVELOPMENT OF THE KALMAN FILTER APPLICATION AND A VHDL MODEL FOR THE AFIT FLOATING POINT APPLICATION SPECIFIC PROCESSOR (FPASP)

THESIS

William E Koch
Capt, USAF

AFIT/GCE/ENG/89D-3

AFIT/GCE/ENG/89D-3

DEVELOPMENT OF THE KALMAN FILTER APPLICATION AND A VHDL MODEL FOR THE AFIT FLOATING POINT APPLICATION SPECIFIC PROCESSOR (FPASP)

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

William E Koch, BSEE

Capt, USAF

December, 1989

Table of Contents

List of Figures

AFIT/GCE/ENG/89D

Abstract

The Air Force Institute of Technology (AFIT) is conducting research that will lead to the development of a Floating Point Application Specific Processor (FPASP). The FPASP architecture is designed around two independent 32 bit data paths that work in tandem to support full IEEE double precision floating point operations, or that can work independently for 32 bit integer processing. Designed to operate at 25 MHz, the FPASP will be capable of performing 25 million floating point operations per second.

A rapid prototyping methodology has been developed for the FPASP. In this methodology, a user identifies an application that could benefit from a VLSI solution. An algorithm of the application is translated into FPASP microcode which can then be programmed into the Laser Programmable Read Only Memory (LPROM) of a blank FPASP. The programmed FPASP can then be mounted on a circuit card and installed in a host system where it would function as a hardware accelerator supporting the user application.

This thesis supports AFIT's continuing development of the FPASP in two areas. In the first part of this thesis, a user application, the Kalman Filter algorithm, is translated into FPASP microcode for programming into the FPASP. To support the microcode generation, verification, and validation of the Kalman Filter microcode program, a software model of the register transfer level design of the FPASP was developed using the C programming language. The successful development of the Kalman Filter microcode program demonstrated not only the FPASP's abilities to implement a complex algorithm, but also the initial phases of the rapid prototyping methodology. In the second part of this thesis, the feasibility of using the VHSIC (Very High Speed Integrated Circuitry) Hardware Design Language (VHDL) to model a complete system is demonstrated by developing a register transfer level model of the FPASP. Using the VHDL model of the FPASP, it was possible to validate the design concepts used in the development of the FPASP, as well as providing support for architectural enhancements by allowing enhancements to be simulated prior to fabrication changes.

DEVELOPMENT OF THE KALMAN FILTER APPLICATION AND A VHDL MODEL FOR THE AFIT FLOATING POINT APPLICATION SPECIFIC PROCESSOR (FPASP)

I. Introduction

1.1 Background

The continuing advances in the field of Very Large Scale Integration (VLSI) circuit fabrication have made it possible to place more circuitry on a single chip. Currently, chips are being manufactured that contain hundreds of thousands of transistors giving the chips the capability to perform functions that just a few short years ago would require a main-frame computer (Sum81). The resulting benefits of this advanced technology is that VLSI architectures can now be applied to a broad spectrum of difficult problems. However, along with new solutions come new problems.

The specification, design, layout, verification, and fabrication of a custom VLSI chip can be a very lengthy and costly process. The cost to design a custom VLSI chip can easily exceed a million dollars, and the lead times required are two or more years. Lengthy lead times and high costs compound the perceived risk of using custom VLSI architectures, causing many program managers to hesitate before committing to a VLSI solution for their processing needs. Another major difficulty with custom VLSI architectures is reliability. As the number of gates on a VLSI chip increases, the testing problem increases exponentially and, unless careful steps are taken during the design phase, it can quickly become impossible to achieve 100% fault coverage (Fuj85).

One approach that will make it easier to introduce VLSI architectures into a broad spectrum of applications is being researched at the Air Force Institute of Technology (AFIT). AFIT is currently developing the hardware and methodologies for rapid prototyping a user programmable, application specific processor. The Floating Point Application Specific Processor (FPASP) currently under development at AFIT will be capable of full IEEE double precision floating point multiplication, addition, and subtraction. In addition to its floating point arithmetic capabilities, the FPASP will provide two mutually exclusive, 32-bit, integer data paths (Com88). Each integer data path will contain separate sets of registers, an arithmetic logic unit (ALU), and a linear shifter giving the FPASP the capability to perform parallel integer computations in any given clock cycle. The FPASP will incorporate a laser programmable read-only rom (LPROM) developed in a thesis by Capt John Tillie, based on research in laser technology at the Massachusetts Institute of Technology Lincoln Laboratory (Til88) (Scr89). The LPROM will enable a user to program the FPASP microcode at some time after fabrication. In this sense, the FPASP can be manufactured *en gross* and individual processors programmed at later dates as specific applications warrant. Hence, a *generic* VLSI architecture that has been thoroughly tested for functionality and reliability can be rapid prototyped and applied to a broad spectrum of computationally intense problems.

Another area of ongoing research at AFIT is in the field of Kalman Filter design. The Kalman Filter algorithm is characterized by the large number of matrix algebra computations it must perform to arrive at an estimate of some values of interest. Additionally, portions of the algorithm require a high degree of numerical accuracy. These characteristics make the Kalman Filter algorithm a likely candidate for implementation with the FPASP. Class projects have developed FPASP microcode for the Kalman Filter. Student suggestions for improving the FPASP architecture to better support matrix algebra computations were incorporated into the final design of the FPASP (Com88) (Lin88). This should not however be construed to imply that the FPASP has been tailored to fit one specific class of problems as will be explained later in this thesis.

1.2 Problem Statement

The design and development of the FPASP architecture has been topic of several theses at AFIT (Gal87) (Com88) (Til88) (Scr89). This thesis further added to the development of the FPASP in three areas. First, the Kalman Filter algorithm was translated into FPASP microcode. The Kalman Filter microcode program will be thoroughly tested prior to release for programming into the FPASP. Second, a means to quickly simulate the microcode was developed. As a functional FPASP is not yet available, this required developing a register transfer level (RTL) model simulator, based on the register level description of the FPASP, using a higher level programming language. And third, the register level specification of the FPASP was modeled using the VHSIC (Very High

Speed Integrated Circuit) Description Language (VHDL) to aid in the validation of the architecture design.

The Kalman Filter algorithm is being supplied by an outside user (AFIT Guidance and Control Group) in the form of a series of linearized mathematical equations, and higher-order language routines (Lew89) (May79). From these equations, high level psuedocode algorithms will be developed that can be mapped into FPASP microcode.

Fundamental to the development of the Kalman Filter microcode is the ability to execute the microcode to test for correctness. An RTL model simulator will be developed using the C programming language that will allow the execution of FPASP microcode. The RTL simulator functionally models all internal register states and all signals that cross internal blocks within the chip. Additionally, the RTL simulator models the FPASP external memory which can be analyzed after a simulation to verify and validate microcode for logic correctness prior to FPASP programming.

Several VLSI tools are available to test FPASP subcomponents at the transistor and gate level. While validation at the component level is a must, *it does not negate the requirement to* validate the design at the register level. VHDL provides the tools necessary to model register level architectures, and simulate those models. In response to the need to validate the FPASP design at the register level, this thesis developed a hardware simulator using the VHDL design language. The VHDL hardware simulator, or the VHDL simulator, provides the means to examine the functionality and interaction of the individual FPASP components from a timing perspective.

At this point, the development of two simulators, both capable of executing microcode, might be questioned. A primitive model of the FPASP written in VHDL has shown that simulating hardware systems can take a great deal of time. The initial study indicated that to simulate the microcode for a 64-state Kalman Filter would require approximately two years to complete which is clearly undesirable. The smallest Kalman Filter, four states, will require a full day to simulate with a hardware simulator. True, VHDL hardware accelerators exist that can decrease simulation times by one or two orders of magnitude, but for the purposes of developing microcode, a decrease of one or two orders of magnitude in simulation time is not sufficient. Hence, the need to simulate developing microcode in a timely manner lead to the need for a simulator that simply tests for logic correctness. Once the microcode has been thoroughly tested with the RTL simulator, the code can be used in the simulated hardware environment with a reasonably good chance for success.

1.3 Scope

This thesis is bounded in both the type of problems to which it applies and to the solutions it presents.

This thesis will examine the overall requirements for developing a microcode applications program for the FPASP to include the development of a VHDL model for system level validation. The development of microcode applications programs includes the requirement for some means to execute the code in a timely manner. Additionally, the author makes no pretense to be an expert in the field of Kalman Filters. Therefore the Kalman Filter algorithm should be completely specified by a high-order language, pseudocode, linearized mathematical formulas, or some combination of the three.

The development of the simulators to test the microcode and validate the system design requires an in depth understanding of timing interrelationships, sub-module functionality, data movement, and overall capabilities and limitations of the FPASP architecture.

1.4 Summary of Current Knowledge

An introduction to developing user applications into FPASP microcode was done as class projects in the EE588 class during the 1988 summer quarter (Lin88). One of the assigned projects was the translation of the Kalman Filter algorithm which was done in its entirety. The Kalman Filter project, along with the other projects was done without the ability to test the microcode. Despite this, many insights into microcode programming were gained in the exercises. In particular, students made suggestions on hardware changes that could enhance the operation of the FPASP, and many of these suggestions were incorporated into the design.

The design of the FPASP evolved from the single precision ASP (SPASP) designed by Capt Gallagher and Capt Linderman. The basic architecture of the FPASP is essentially two 32–bit data paths working in tandem with double precision floating point macrocells, or working separately to perform integer operations and software support such as looping and branching (Com88). The

FPASP is designed using a bit slice methodology. Each microword consists of 128 bits divided into upper and lower rom words. There are 41 instruction fields within each microword. An efficient translation of a user algorithm will require effective utilization of each instruction field. Insight gained from the EE588 class projects showed that the limiting factor in translating user algorithms was the efficient utilization of the FPASP's floating point hardware. As such, algorithms may have to be completely redesigned from the perspective that a sequential approach to implementing an algorithm may not be the best approach in the internal parallel environment of the FPASP architecture.

1.5 Assumptions

The translation of the Kalman Filter algorithm and the development of both the logic and hardware simulators is contingent on the stability of the FPASP microword format and the microcode field definitions. The format and definitions of the microcode were developed and finalized in Capt John Comtois' thesis and the success of this thesis effort depends strongly on maintaining the current microword configuration. Cursory reviews of the Kalman Filter algorithm indicate the microcode for the algorithm will be quite extensive. Any change in the microword format at this stage of development could, and probably will, render the microcode unusable. The impact of changed microword formats will not impact the logic and hardware simulators to the same degree. Therefore, if changes to the microword format are deemed necessary, these changes will have to carefully coordinated to ensure the success of this research program.

This thesis also assumes that the register level design of the FPASP is complete and finalized. Changes in the FPASP design at the register level will directly effect the usefulness of both the RTL simulator and the VHDL simulator.

1.6 Approach

The primary goal of this thesis is successful development of the Kalman Filter microcode program and the development of a VHDL model of the FPASP at the register level. Presently, no means exists to execute FPASP microcode necessarily implying microcode development can not be successfully effected until a software model of the FPASP is developed. And again, a reliable software model of the FPASP can not be developed until the register level design of the architecture is completely understood. An understanding of the architecture is also required for the development of the VHDL hardware simulator.

This thesis will use a tiered approach starting with a thorough study of the FPASP architecture itself. Particular attention will be paid to the movement of data throughout the FPASP and to external memory. Next, the internal timing of the FPASP will be studied to gain an understanding of the sequencing of internal operations and data movement.

After gaining an understanding of the internal operations and timing of the FPASP, the design and development of the simulators can begin. The RTL simulator will address only the interpretation of the microword and the logical flow of data through the FPASP. Any timing considerations will deal only with the fact that certain operations must occur before others leaving the magnitude of time delay for the VHDL simulator.

Once the RTL simulator been completed and sufficiently tested to give a reasonable degree of confidence, the development of the Kalman Filter algorithm can begin. Previous experiences with the EE588 class projects have shown that FPASP microcode development is an iterative process concerned with efficiently using the FPASP resources and keeping the word count down.

After completing the Kalman Filter microcode program, the development of the VHDL simulator can begin. The VHDL simulator will develop models of each of the FPASP components, and particular attention will be given to the inter-module and intra- module timing requirements.

1.7 Materials and Equipment

The development of the FPASP simulators will involve production of VHDL simulation models. Therefore, a VHDL analyzer and simulator will be required. The VHDL environment runs on the VAX/VMS and SUN/UNIX computer systems. VHDL 1076, version 2.0, is currently available on the AFIT VAX/VMS system and the AFIT SUN 3 systems operating under the UNIX V4.2 operating systems. Additionally, access to an AFIT SUN 3 workstation will be required.

1.8 Sequence of Presentation

Chapter 1 has provided background into the development and projected capabilities of the FPASP. The chapter has gone into some detail analyzing the problems this thesis proposes address, and the approaches that will be used.

Chapter 2 will present an analysis of the problems and requirements which effect the development of the application development environment as well as the problems and requirements of translating the Kalman Filter algorithm into FPASP microcode.

Chapter 3 will review and analyze the FPASP architecture to include a breakout of th individual sub-modules, sub-module interconnectivity, data transfer timing, and the FPASP microword and its associated microfield definitions.

Chapter 4 will detail the design and development of the C RTL simulator for the FPASP. The chapter will also cover the limitations of the RTL simulator, as well as its benefits.

Chapter 5 will detail the design and development of the VHDL simulator for the FPASP. The chapter will discuss each of the design entities developed to model the FPASP.

Chapter 6 will detail the translation of the Kalman Filter algorithm into FPASP microcode, and will include explanations of how the validation and verification of the microcode was performed.

Chapter 7 will present conclusions related to the development of the Kalman Filter application, and the VHDL simulator. Also, recommendations are made for follow on studies.

II. Problem Analysis

2.1 Introduction

The current state of integrated circuit technology allows the application of VLSI architectures to a broad range of difficult problems. Many research efforts throughout the Department of Defense (DoD) require special purpose processors and controllers, however the long lead times and high costs have tempered the application of advanced VLSI architectures (Sum81). AFIT recognizes this problem and has proposed an alternate approach to developing a specific architecture for a specific application. The approach proposed by AFIT is the development of a user programmable, application specific, processor (Gal87) (Com88). AFIT has undertaken a research program that will provide the methodologies to rapid prototype application specific processors which will culminate with the development of the FPASP.

Currently, the register level design of the FPASP architecture is complete, and the final design and testing of the individual macro-cells is nearing completion. This chapter will review the overall FPASP rapid prototyping scenario, and discuss and analyze the requirements for developing and testing FPASP microcode.

2.2 FPASP Applicability

The FPASP has been optimized to support computationally intensive applications. Applications that cannot be specified algorithmically, such as word processors, or which have special hardware requirements beyond those provided by the FPASP architecture may not be good candidates for this rapid prototyping methodology. The FPASP has been specifically designed to support matrix algebra in which the most basic operation is the dot product of two vectors. To support an effecient dot product routine, the hardware design of the FPASP architecure has been refined, as results of EE588 class projects feedback, to perform the basic operations of multiplication and addition in as few clock cycles as possible.

The optimized design of the FPASP architecture to allow efficient matrix algebra operations does not limit the usefulness of the FPASP to a particular subset of applications. EE588 class projects effectively developed microcode for a variety of applications. The class projects that were successfully completed are: Kalman Filter propagation and update routines; Neural Network Training, where forward and backward propagation algorithms were translated into microcode; LMS Adaptive Filtering, which includes a microcode version of the dot product routine applicable for complex data vectors; LOG base 2 and ArcTan(), which used the Chebychev polynomials was translated into microcode; and Singular Value Decomposition (SVD) translated a Fortran routine for SVD into microcode. The EE588 class projects demonstrated the versatility and broad range of applications the FPASP is well suited for.

2.3 FPASP Rapid Prototyping

The rapid prototyping methodology for the FPASP has been designed to make it easy insert the FPASP into wide range of applications. After a user has identified an application that could benifit from the capabilities the FPASP offers, an algorithm of the application is translated into FPASP microcode. After the algorithm is translated into FPASP microcode, the microcode is verified and validated, and then programmed into the LPROM. The FPASP is then mounted on a circuit board, tested, and interfaced into the user's application.

It is expected that microcode development will consume the majority of the time required to prototype the FPASP. There are two ROMs into which the microcode will be written. The first ROM is the LPROM which has already been discussed, and the second ROM is a fixed ROM which will contain a library of microcode subroutines. The microcode subroutines that will be placed in the fixed ROM during manufacture are matrix multiplication, matrix addition, square root, and Newton Rasphon Inversion. The use of these fixed subroutines should ease the translation of a user algorithm into FPASP microcode.

2.4 Microcode Development

A user algorithm, whether it is in a higher–order language form, pseudocode, or linearized mathmetical formulas, tends to be sequential in nature in that one subset of the algorithm is performed in its entirety before progrssing onto next subset. While a straight translation of such an algorithm into microcode is possible, such a translation would in all likelyhood be inefficient. As previously noted, the FPASP microword is comprised of 41 individual microfields allowing for the parallel operation of many internal FPASP functions. Translating a user algorithm into microcode must take this capability into consideration if the resultant translation is to effectively and efficiently utilize the FPASP to its fullest extent. Parallelizing the internal operations of the FPASP does not imply the possibility of subdividing the user algorithm into individual tasks that can process concurrently. Rather, parallelizing FPASP internal operations implies that while one portion of the algorithm is being processed by the FPASP, hardware not currently involved with the current processing is working to intialize registers, ALUs, shifters, etc. for the next portion of the algorithm to be executed.

For example, the current portion of the user algorithm being executed by the FPASP is a loop that will perform n iterations. In each iteration of the loop, a subroutine will be called to calculate the dot product of two vectors, after which the dot product is scalar multiplied and written out to memory. The dot product routine requires several general purpose registers, pointers, and incrementers to be preset prior to the subroutine call. The microcode could be written in a purely sequential fashion to first intitialize the necessary registers, pointers, and incrementers, call the dot product routine, perform the scalar multiplication, and write the result out to memory. In this approach, several lines of microcode would be required solely to initialize resisters and pointers. Assuming it requires x lines of microcode to perform the intializations, this method of translation would add an additional xn clock cycles of overhead to the algorithm. A more efficient alternative, which emphasizes the concurrent abilities of the FPASP, would be to intialize the registers, pointers, and incrementers for the first call to the dot product routine prior to entering the loop, and performing subsequent initializations concurrently with the scalar multiplication and memory I/O. This approach at worst will add only x clock cycles for the first intitialization, yet will save $x(n-1)$ clock cycles incured in the previous serialized translation. An even more judicious approach might find it possible to perform the first initialization concurrently with the operation leading up to the loop eliminating the need for the x extra clock cycles.

Equally important in developing efficient microcode programs is the full utilization of the FPASP's dual integer data paths. Manipulation of matrix data structures within the FPASP requires numerous integer computations which are used for indexing, and column and row counters. While one data path is computing the number of elements within a matrix data structure, the other data path can concurrently compute the distance between the row or column elements of the same structure.

2.5 Microcode Verification and Validation

The previous section highlighted some of the requirements for developing efficient microcode. An important step in developing microcode is the ability to verify portions of the code as it is being generated. There are two distinct aspects of the microcode or submodules of the microcode that need verification. The first is verification of logic correctness. Logic correctness ensures all general purpose registers, pointers, loop counters, and incrementers have been properly intialized when they need to be intitialized, and that the routine executes properly. The RTL simulator discussed in Chapter 1 will be able to perform this verification. The second is verification that internal FPASP timing requirements have been met. The internal operations of the FPASP have been designed to occur in one clock cycle. The exceptions are the operations of the floating point multiplier and floating point adder. Both these circuits have a latency of three clock cycles with a two cycle execution. Additionally, flags set as a result of ALU and shifter operations incur a one clock cycle delay and are therefore valid in the clock cycle following an operation. Flags associated with the floating point hardware aren't set until the data is driven out of the floating point hardware, and, as with the ALUs and shifters, can't be read until the following clock cycle. The VHDL hardware simulator will have the ability to model the FPASP timing element, and therefore will provide the means to verify the timing aspects of microcode applications.

Validation of microcode entails testing to ensure the developed microcode does in fact implement the user algorithm it was designed for. Validation therefore necessarily implies simulating the microcode routine with a user supplied data set, and comparing the simulated results against known valid results. With the Kalman Filter algorithm, this will necessitate being able to simulate the microcode with fairly large data sets, and performing the simulation a number of times to propagate and update the data. A final review of the output data will insure that the microcode implementation of the Kalman Filter is correctly performing the time propagations and measurement updates correctly, and not causing the data to diverge from a user supplied truth model.

Kalman Filter FPASP Performance					
N	M	S	P	X	ΔT
states	measurements	noises	controls	terms	msecs
64	32	32	16	1	502
64	32	32	16	2	528
40	20	20	0	1	125
40	20	20	0	2	131
20	6	6	0	1	15
20	6	6	0	2	16
16	1	6	0	1	6.9
16	1	6	0	2	7.4

Table 2.1. EE588 Kalman Filter Class Project Results

2.6 Kalman Filter Microcode Translation

The main benifits of implementing the Kalman Filter with the FPASP is speed of computation and precision of calculations. The high precision of calculations is gained by using a processor capable of double precision IEEE floating point operations. Speed of computation is dependent on not only the speed of the hardware, but also on the *cleverness* of the microcode development as discussed above. Although no specific requirements for execution speeds have been levied on this research project, the quicker the FPASP can perform the Kalman Filter time propagations and measurements updates, the more dynamic the environment the system can operate in.

The EE588 Kalman Filter class project provided some insight into the expected performance of the FPASP when applied to the Kalman Filter problem, and the results are given in Table 2.1 (Hus88). The times given are for a complete propagation and update of the Kalman Filter. These times are however far from optimal. Referencing the first table entry, the total number of clock cycles required for a complete propagation and update are 12,543,132, and the total number of floating point operations are 3,529,726 which yields 7.04 MFlops/sec out of a maximum 25MFlops/sec. Optimization of the Kalman Filter microcode will require a complete restructuring of the current microcode, and for this reason, it was deemed appropriate to scrap the microcode and rewrite it. An additional factor which weighed heavily in the decision to scrap the original microcode was the fact that the microcode was written using the original (now outdated) version of the microcode.

2.7 *Simulator Development*

The development of the FPASP simulators approach the architecture from two distinct view points. The logic simulator views the FPASP architecture as a collection of functions that can be modeled using subroutines. Execution of any given function during a simulation cycle is dependent on the ROM word micro–field controlling that function, and the current phase of the simulation cycle. This implies that subroutines developed to emulate the functions of the FPASP will have to be sequenced in the correct order according to when in the simulation cycle they must occur. The VHDL hardware simulator will model the FPASP in an entirly different way. *Design entities* will be designed and developed that model the behavior of each FPASP component at the register level. Each design entity will then be invoked concurrently according to an internal clock that models the FPASP's external clock. The execution a design entity is also dependent on the ROM word micro–fields governing the particular design entity. In both cases, the development of the simulators is highly dependent on understanding the internal timing of the FPASP in addition to understanding the functions of the individual components.

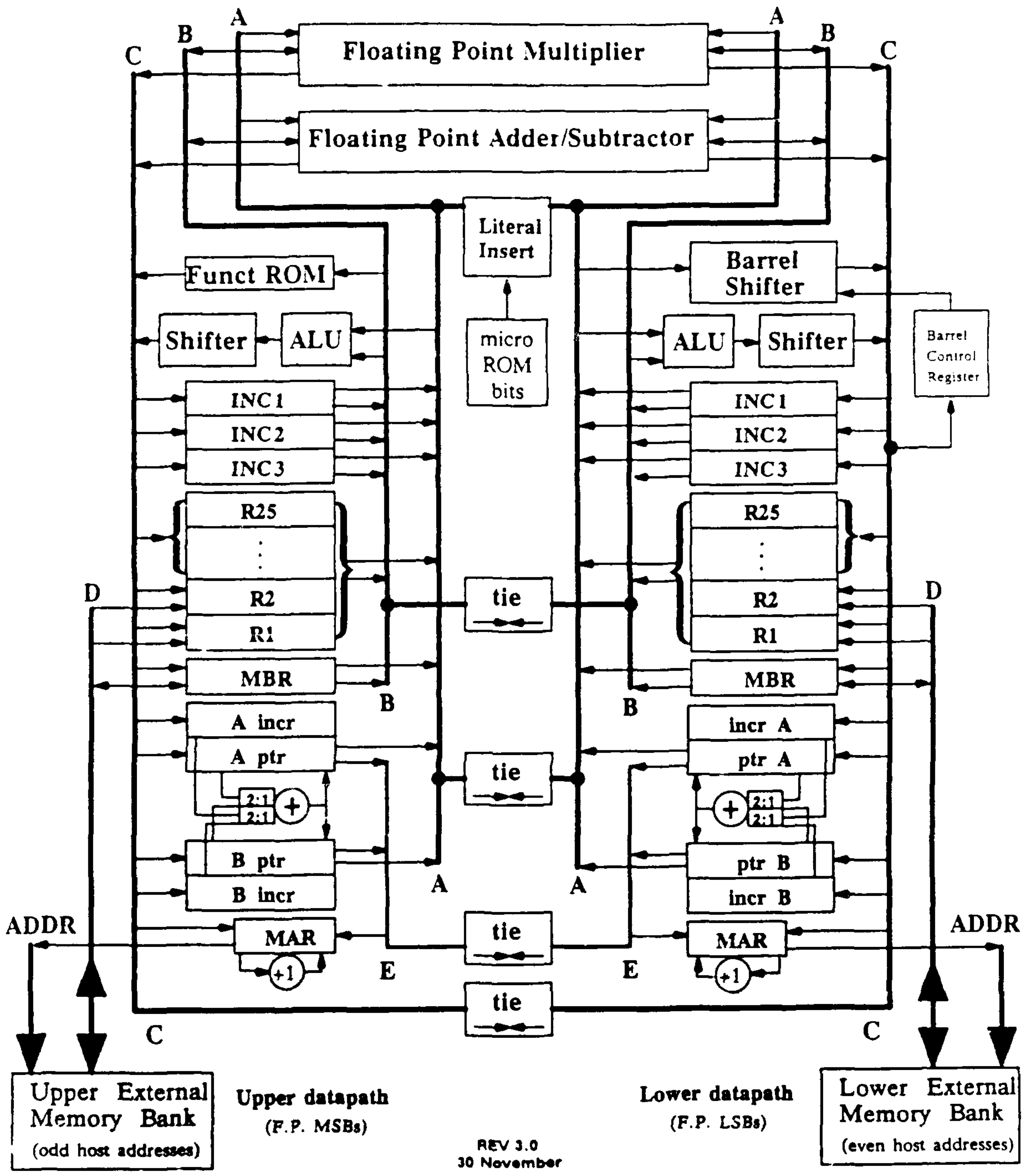
III. The FPASP Architecture

3.1 Introduction

Prior to any development of the FPASP logic and hardware simulators, and the development of any microcode, a complete understanding of the FPASP architecture is needed. This chapter examines the architecture and internal timing of the FPASP. The chapter will start by providing an overview of the FPASP architecture and will then examine the individual modules which go together to comprise the overall FPASP.

FPASP Architecture Overview: The FPASP is a laser programmable VLSI processor which will interface to a host computer system. The data paths shown in Figure 3.1 depicts the FPASP as two tightly coupled 32 bit integer processors which together feed the floating point macrocells that perform IEEE standard double precision floating point multiplication and addition/subtraction operations. Special floating point hardware allows these operations to be performed with a latency of three clock cycles (load, compute, drive) but with a throughput of two clock cycles (loading and driving may be overlapped).

The 32 bit integer data paths are mirror images of each other with a few exceptions. Each has a 32 bit integer arithmetic logic unit (ALU) and linear shifter which can perform independently of one another, 25 general purpose registers which can store intermediate integer results or half of a floating point operation result, three incrementable registers with auto–increment capability to hold loop counter variables, one memory buffer register (MBR) which connects through a D Bus to its respective memory bank, two external memory pointer registers (A,B,C, and D ptrs) with associated increment registers (A,B,C, and D INCR) that can be preset with a desired increment amount, and one memory address register (MAR) with auto–increment capability. The upper 32 bit data path contains a Function ROM which is programmed with fixed values used as seeds for iterative routines. The lower 32 bit data path contains a Barrel shifter for shifts longer than one bit. Common to both data paths is the Literal Inserter which allows for the direct insertion of variables from the microcode onto the busses. Finally, all of the data busses are interconnected through Bus Ties that allow values driven on one bus to be driven onto their counter part busses.



Floating Point Multiplier
Floating Point Adder/Subtractor
A
B
C
Literal Insert
micro ROM bits
Funct ROM
Shifter
ALU
Barrel Shifter
Barrel Control Register
INC1
INC2
INC3
R25
R2
R1
MBR
A incr
A ptr
B ptr
B incr
incr A
ptr A
ptr B
incr B
2:1
+
+1
MAR
tie
D
E
ADDR
Upper External Memory Bank (odd host addresses)
Lower External Memory Bank (even host addresses)
Upper datapath (F.P. MSBs)
Lower datapath (F.P. LSBs)
REV 3.0
30 November

Figure 3.1. FPASP Register-Level Description

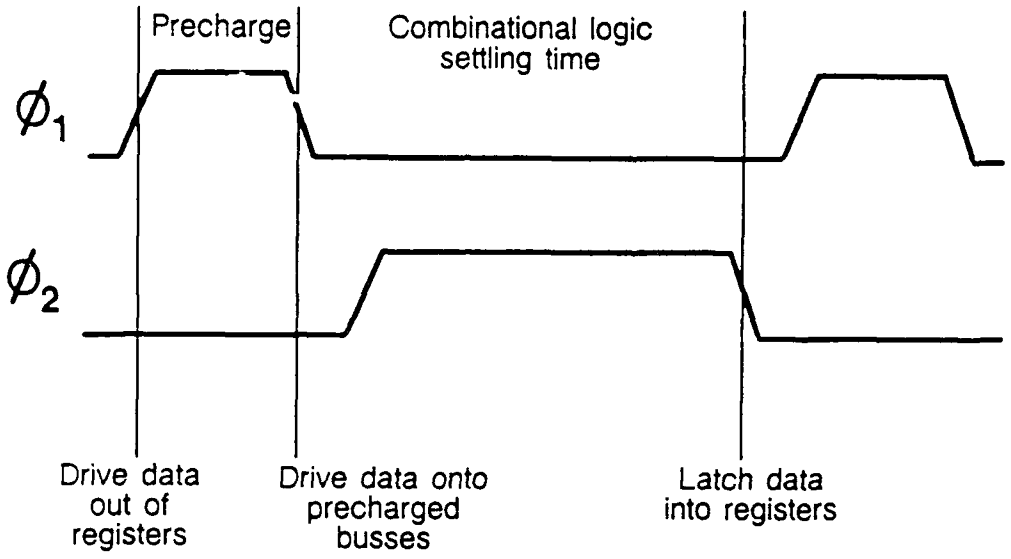

Precharge
Combinational logic settling time
ϕ_1
ϕ_2
Drive data out of registers
Drive data onto precharged busses
Latch data into registers

Figure 3.2. FPASP Clock Cycle

Controlling the operation of the FPASP and not shown in Figure 3.1 is the microsequencer. A full discussion of the microsequencer will be provided later in this chapter.

The following sections will describe in some detail the operation of each of the individual sub-components that go together to make the FPASP. Prior to any in–depth discussion on component structure and characteristics, it is first necessary to discuss the timing of data transfers within the FPASP.

3.2 FPASP Clock

The FPASP is designed for a clock speed of 25Mhz with its corresponding period of 40ns. As shown in Figure 3.2, the clock is comprised of two non–overlapping clock signals Φ_1 and Φ_2.

The FPASP uses master–slave (MS) flip–flops throughout its architecture to prevent the condition known as data racing. Data racing occurs when a **D** flip–flop is enabled and before its data input can be disabled, the input changes. As a consequence, the latched value is not the correct

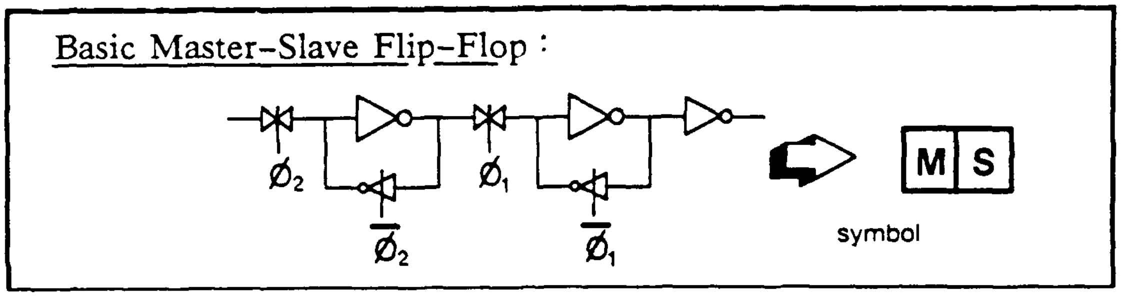


Figure 3.3. Typical FPASP MS Flip-Flop

value. MS flip–flops use two ganged **D** flip–flops to trap the data by enabling each **D** flip–flop at different times. A typical MS flip–flop that is used in the FPASP is depicted in Figure 3.3.

With the rising edge of Φ_2, the input flip–flop is enabled and the output of the input flip–flop is free to follow the changing conditions on the bus. When Φ_2 falls, the input flip–flop is disabled latching whatever value was on the bus at that time. A short time later, the rising edge of Φ_1 enables the output flip–flop and the latched value in the input flip–flop can now be sensed at the output of the register. The output of the register is latched with the falling edge of Φ_1.

3.3 FPASP Register Types

There are five distinct register types used in the FPASP all of which use the MS flip–flop configuration described above. The different register types listed in the order they will be discussed are: the general purpose registers; the incrementable registers; the pointer registers; the memory buffer registers; and the memory address registers.

General Purpose Registers: Both the upper and lower data paths contain 25 general purpose registers. Each general purpose register is made up of 32 individual MS flip–flop macro–cells corresponding to the 32 bit–width of the busses. Data stored in the general purpose registers are driven onto precharged **A** or **B** busses with the rising edge of Φ_1. New data is latched into the general purpose registers from the **C** bus with the falling edge of Φ_2.

There are two micro–fields within the microword for each data path controlling which register is driving the **A** bus and which register is driving the **B** bus. It is therefore possible to have the same register drive both busses. There is also a single micro–field for each data path to determine which register is loaded from the C bus. Only one register can be loaded at a time.

Incrementable Registers: Each data path has three incrementable registers specifically designed to hold loop counter variables. Each incrementable register has the associated hardware necessary to give them an auto–increment capability. Additionally, each incrementable register has a zero flag that can be tested the clock cycle after an increment instruction to determine if the register has incremented to zero. To control the number of iterations through a loop, an incrementable register is loaded with the negative number of times the loop is to be traversed. The incrementable register is incremented with each iteration through the loop and the loop is exited when the appropriate zero flag is set.

There is a single micro–field within the microword for each data path which controls incrementing the registers. Control of loading the incrementable registers is contained within the same micro–fields that control loading the general purpose registers.

Pointer Registers: The pointer registers are designed to directly support addressing the individual elements of a matrix. Two dimensional matrices are generally stored in memory in row–major format. To address successive column elements of a matrix is a simple matter of incrementing a memory pointer by one. However, to address successive row elements of a matrix, it is necessary to increment the memory pointer by the row dimension of the matrix. Each data path has two independent pointer registers, and each pointer register has an associated increment register and adder to make the increment. The pointer register is loaded with an initial value and its associated increment register is loaded with an increment value (positive or negative). With every microcode instruction encountered to increment the pointer, the pointer value is incremented (or decremented) by the value contained in its increment register.

Loading the pointer registers is again controlled by the same micro–field as the general purpose registers. However, there is a dedicated micro–field for each data path to control loading the associated increment registers and the incrementing of the pointers. Both the pointers and their increment registers can be loaded at the same time. However, since there is only one C bus for each data path, unless it is desired to load the same value, separate clock cycles are needed to load different values.

Memory Buffer Registers: The memory buffer registers (MBRs) are similar to the general purpose registers. The MBRs drive data to the data pads of the FPASP via the **D** bus. Originally intended to be the only registers to read data in through the data pads, the role of the MBRs was extended to include both the upper and lower general purpose registers R1 and R2. Writing data out through the data pads requires writing the data into the MBRs and lowering the Write Enable signal lines.

There are four micro–fields that control the flow of data through the MBRs. The same three micro–fields that control driving out of the general purpose registers onto the **A** or **B** bus and driving data into the general purpose registers from the **C** bus also control the MBRs in the same fashion. There is an additional micro–field that controls driving data out of the MBRs onto the **D** bus.

Memory Address Registers: Addressing of external memory is controlled by a memory address register (MAR) in each data path. Each MAR has an auto–increment capability. The normal source for loading address values into the MARs is from the pointer registers via the **E** bus. However, the MARs are also connected to the **C** bus allowing any register to hold an address.

There is single micro–field for each MAR that controls whether the MARs are loaded from the **E** bus or the **C** bus or if their present values are incremented by one.

3.4 FPASP Processing Components

The FPASP has seven processing components working together that support three types of computation: logic operations, 32 bit arithmetic, and IEEE standard double precision floating point arithmetic. In addition to the above computations, the FPASP also supports a 32 bit integer multiplication option using the floating point multiplier. The processing components that will be discussed in this section are: the integer arithmetic and logic units (ALUs), the linear shifters, the barrel shifter, the literal inserter, the function ROM, the floating point multiplier, and the floating point adder/subtractor.

FPASP ALU Logic Operations		
Function	Value Passed to Shifter	Flags Affected CARRY,OVERFLOW,SIGN,ZERO
MOVN	A	none (default)
OR	A OR B	ZERO
AND	A AND B	ZERO
XOR	A XOR B	ZERO
MOV	A	ZERO
NAND	A NAND B	ZERO
NOR	A NOR B	ZERO
NOT	NOT A	ZERO
INC	A + 0 + 1	All Four
SET	A + B + 1	All Four, Sets Carry
ADC	A + B + previous carry	All Four
ADD	A + B + 0	All Four
NEGA	$\bar{A}$ + B + 0	All Four
SUB	A + $\bar{B}$ + 1	All Four
SWB	A + $\bar{B}$ + previous borrow	All Four
DEC	A + $\bar{1}$ + 0	All Four

Table 3.1. FPASP ALU Operations

Integer Arithmetic and Logic Units (ALUs): The FPASP has two independent ALUs, one on each data path. The ALU functions supported by the FPASP are listed in Table 3.1.

As illustrated in Table 3.1, the two ALUs generate separate flags which are stored in MS flip–flops that can be used as branch conditions by the ı :rosequencer. The MS flip–flops add a one clock cycle delay from the time the flag is generated until it can be used by the microsequencer for branching instructions. These flags as well as the interconnectivity of the individual components that comprise the ALUs are shown in the block diagram in Figure 3.4.

Bidirectional Linear Shifter: The bidirectional linear shifter is depicted in Figure 3.4. The result of an ALU operation is fed directly through the shifter to the **C** bus. The linear shifter can perform one bit shifts in either direction. The type of shifts that the linear shifter can perform are listed in Table 3.2.

The linear shifter is capable of performing arithmetic shifts, logical shifts, circular shifts, and shifts using bits stored from previous ALU operations. The bit shifted out is saved in a MS

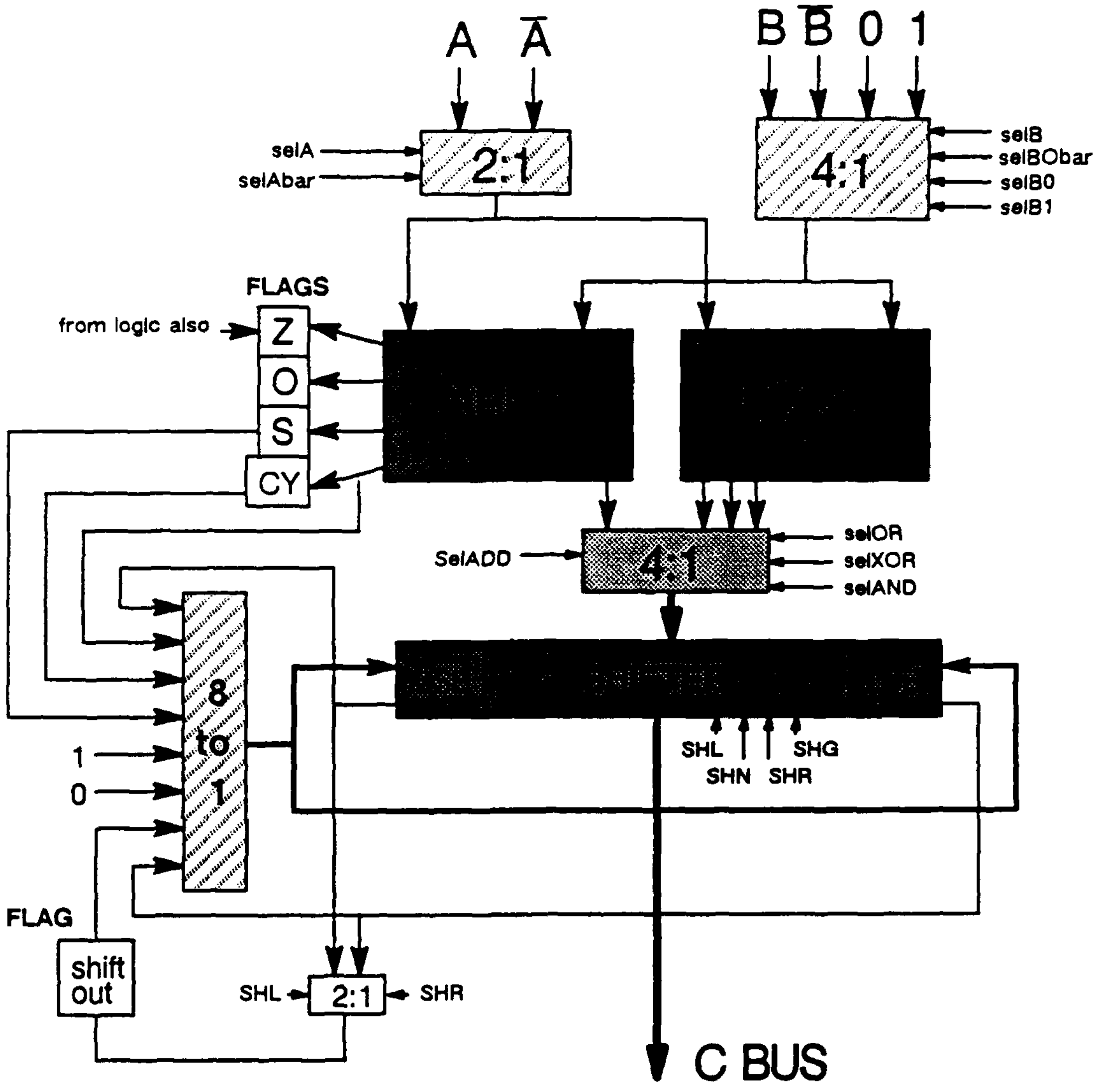


Figure 3.4. FPASP ALU and Linear Shifter

Bidirectional Linear Shifter Operations		
Function	Type of Shift	Bit Shifted In
NOP	Shifter does not drive C bus	none
GNDC	Shifter grounds C bus	none
PASS	No shift, ALU output to C bus	none
SLOT	Chained left shift	Previous shift–out bit into LSB
SLMS	Circular left shift	MSB circulated into LSB
SLCY	Shift left with CARRY	CARRY of present ALU operation
SL0	Shift left with zero	0 into LSB
SL1	Shift left with one	1 into LSB
SRLS	Circular right shift	LSB circulated into MSB
SRCF	Shift right with previous CARRY	CARRY flag into MSB
SRS	Shift right with previous SIGN	SIGN flag into MSB
SROT	Chained right shift	Previous shift–out bit into MSB
SRSE	Arithmetic right shift	MSB extended
SRCY	Shift right with CARRY	CARRY of present ALU operation
SR0	Shift right with zero	0 into MSB
SR1	Shift right with one	1 into MSB

Table 3.2. FPASP Linear Shifter Operations

flip–flop, and can be chosen as the shifted in bit at the next clock cycle or as a condition flag by the microsequencer during the next clock cycle. The chained interconnection of the ALU and the linear shifter places a dependency on the independent operations the two components. To use the ALU and have its results driven out onto the **C** bus, the linear shifter PASS function must be selected during the same clock cycle as the ALU function is being executed. Likewise, the ALU must not be selected (the default is MOV which passes the contents on the **A** bus through the ALU to the shifter) during the same clock cycle the shifter is functioning.

Barrel Shifter: The Barrel Shifter was designed into the FPASP to facilitate left circular integer shifts greater than one bit. The number of bit positions the Barrel Shifter can shift an input variable during one clock cycle can be selected from 1 to 31 bits from two different sources.

One source that can be used to control the number of shifts the Barrel Shifter performs is the Barrel Shifter control register. The control register for the Barrel Shifter can be loaded directly from the microcode, or from the five least significant bits (LSB) of the **C** bus in the lower data path. Taking the control input directly from the **C** bus provides a means to control the length of the shift based on a previous calculation. The Barrel Shifter control register is a five bit MS flip–flop and therefore the control register must be loaded the clock cycle prior to the shift operation.

The Barrel Shifter can be made to perform linear shifts. A mask is inserted onto the lower data path **A** bus using the literal inserter (explained below) at the same time the integer to be shifted is driven onto the bus. The mask inserts zeros on top of the integer bits that would be circularly shifted back into the LSB with the results effectively mimicking a left linear shift. The entire operation as before can be performed in one clock cycle. Unlike the linear shifter discussed above, the Barrel Shifter cannot pass its input unchanged through to the **C** bus.

Literal Inserter: The literal inserter provides the FPASP with the capability to insert data onto either the upper **A** bus or the lower **A** bus or both directly from the microcode. This effectively allows integer constants to be stored imbedded in the microcode rather than in memory. The size of the microcode literal field is 16 bits allowing only half of a 32 bit word to be introduced at a time. The versatility of the literal inserter allows those 16 bits to be inserted in the lower half of the word while zeroing out the upper half, or inserted in the lower half while leaving the upper half unchanged, or inserted in the upper half zeroing out the lower half, or inserted in the upper half while leaving the lower half unchanged.

Function ROM: The function ROM consists of seven pages of laser programmable ROM with each page consisting of 32, 5–bit words. The purpose of the function ROM is to provide fixed storage for seed values that is user programmable. The block diagram of the function ROM given in Figure 3.5 shows the ROM takes three control bits from the microcode selecting one of seven memory pages, and five bits from the datapath to index into the chosen memory page.

The function ROM directly supports recursive floating point routines requiring seed values such as inversion or square root routines that are based on the Newton–Raphson Convergence Theorem. Referring to Figure 3.5, a page of seed values is selected using three control bits from the microcode word. The particular seed value within the page is selected using the four MSBs of the mantissa and LSB of the exponent of the number that is being operated on. The output of the Function ROM is placed directly on the **C** bus and consists of the four MSBs of the mantissa and the LSB of the exponent of the seed value.

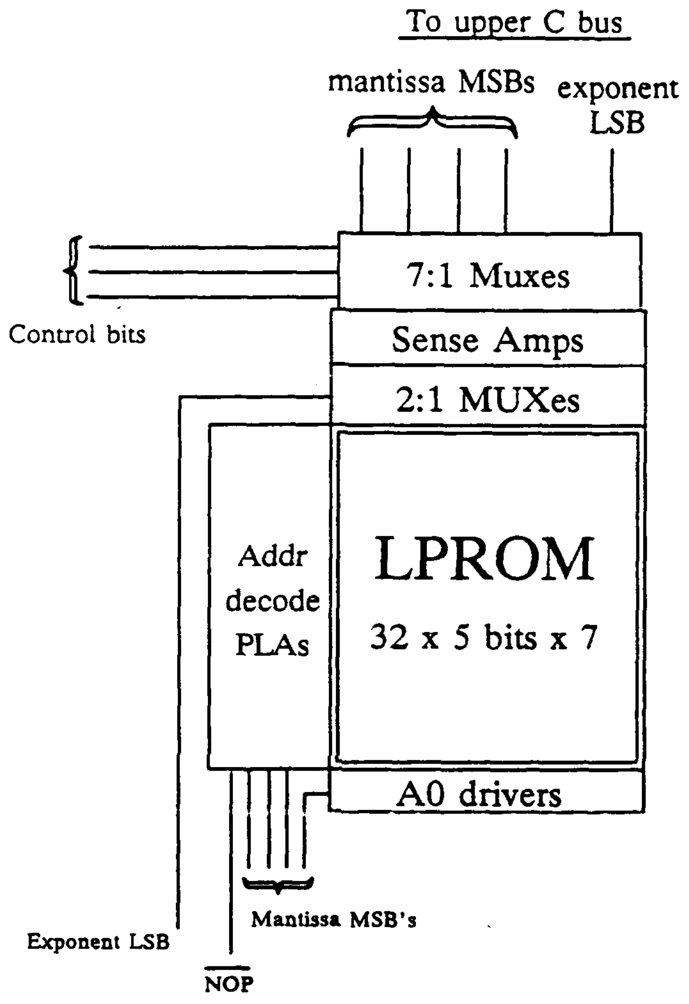


Figure 3.5. FPASP Function ROM

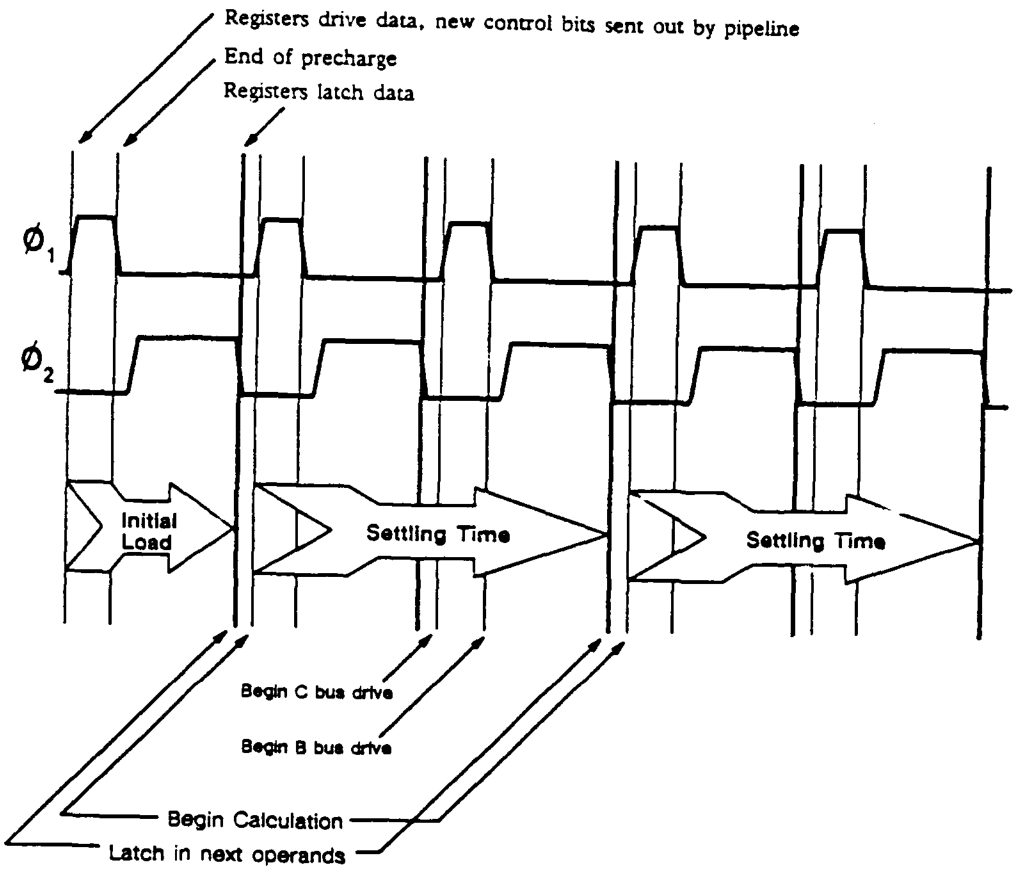


Figure 3.6. Floating Point Hardware Timing

FPASP Floating Point Multiplier: The FPASP floating point multiplier is a single combinational logic circuit designed to implement the octal Booth's encoding scheme. The gate delays through long combinational paths requires that the multiplier be given two clock cycles for settling time as shown in Figure 3.6.

The floating point multiplier supports multiplication of IEEE standard floating point numbers. This numbering notation uses 64 bits where the lower 52 bits represent the mantissa of the number, bits 63 downto 52 are used to represent the exponent, and bit 64 is the sign bit. The FPASP manipulates this 64 bit number representation by splitting the number in half with the lower 32 bits on the lower data path and the upper 32 bits on the upper data path. Loading the floating point multipliers input registers requires the use of both upper and lower A and B busses. Once the input registers are loaded, the computation of the floating point product proceeds by default unless an integer multiply has been explicitly requested. Two clock cycles later, a valid result is available at the output of the multiplier. The output of the multiplier can be driven either onto the **C** busses or onto the **B** busses. The ability to drive the results onto the **B** busses gives the FPASP the capability to load the multiplier or the floating point adder/subtractor on the clock cycle the result is valid versus the intermediate step that would be required to drive the results into a register from the **C** bus and from the register to the **B** bus.

The floating point multiplier supports the IEEE double precision floating point format that requires the multiplier to flag abnormal conditions. The conditions flagged by the multiplier are underflow, zero result, denormalized result, overflow, and a result that is not a number. As with all the flags discussed so far, these flags are available to the microsequencer so that conditional branches can be made to exception handlers. In addition to the single flags generated by the multiplier, the overflow and not–a–number flags are OR'd together along with the overflow and not–a–number flags from the floating point adder and the upper and lower alu overflow flags into a single flag called TRPS. This flag is also available to the microsequencer for conditional branching.

The floating point multiplier also performs 32 bit integer multiplication. The integers to be multiplied are driven into the multiplier from the lower **A** and **B** busses. The result, if it is less than 32 bits can be driven out onto the lower **C** and or **B** busses. If the result is greater than 32 bits, the 32 LSBs are driven onto the lower **C** and or **B** busses as before, and the 32 MSBs are driven onto upper **C** and or **B** busses and the overflow flag is raised.

FPASP Floating Point Adder/Subtractor: The floating point adder/subtractor, as the floating point multiplier, supports the addition or subtraction of IEEE standard double precision floating point numbers. The interface to the floating point adder/subtractor is identical to the floating point multiplier. The microword contains a dedicated microfield for the control of the floating point adder/subtractor enabling the adder/subtractor to operate independent of the multiplier. Similar to the operation of the multiplier, once the adder/subtractor input registers are loaded, floating point addition occurs as the default. To perform floating point subtraction, the subtractor must be selected at the time of the input registers initialization, and then on the following clock cycle, an implicit subtraction must be requested.

3.5 FPASP Microsequencer

The internal sequencing of events of the FPASP is controlled by a microcode sequencer. The major components of the microsequencer are depicted in the block diagram shown in Figure 3.7 and consist of the pipeline registers, the microcode ROMs, the microaddress stack, and the condition multiplexer. This section will examine each of these components.

Microinstruction Pipeline: The storage of microcode in the FPASP is done in two seperate ROMs as previously discussed. There is the fixed ROM which stores microcode subroutines designed to be called by user microcode routines that were programed into the LPROM. The outputs of the ROMs are driven into the inputs of the pipeline registers on one clock cycle and driven out of the pipeline registers to the datapath hardware on the next clock cycle. The one clock cycle ensures the control bits passed to the datapath hardware are stable for the entire length of the clock cycle allowing sufficient time for decoding. The control bits are driven out to the datapath hardware with the rising edge of Φ_1.

Microcode ROMs: The purpose of the microcode ROMs, fixed ROM and LPROM, have already been discussed in some detail. As illustrated in Figure 3.8 these two individual ROMs share a contiguous address space and are treated as a single microcode ROM by the microsequencer.

The microcode memory is organized as 784 words deep by 128 bits wide. There are two halve to each microcode ROM, each one 64 bits wide. There are 640 words of fixed ROM and 144 words of LPROM. The address lines to the two ROMs are decoded to determine which microcode word is read and from which ROM it is read from.

Microaddress Stack: The FPASP is supplied with a microaddress stack allowing subroutine and recursive subroutine calls. The stack has enough registers to support nested subroutine calls 16 deep. As shown in Figure 3.9, should an address be pushed onto the stack when the stack is full, the bottom of the stack is pushed out into external memory starting at memory location zero. The stack pointer generates the addresses used to read and write the internal stack contents from external memory.

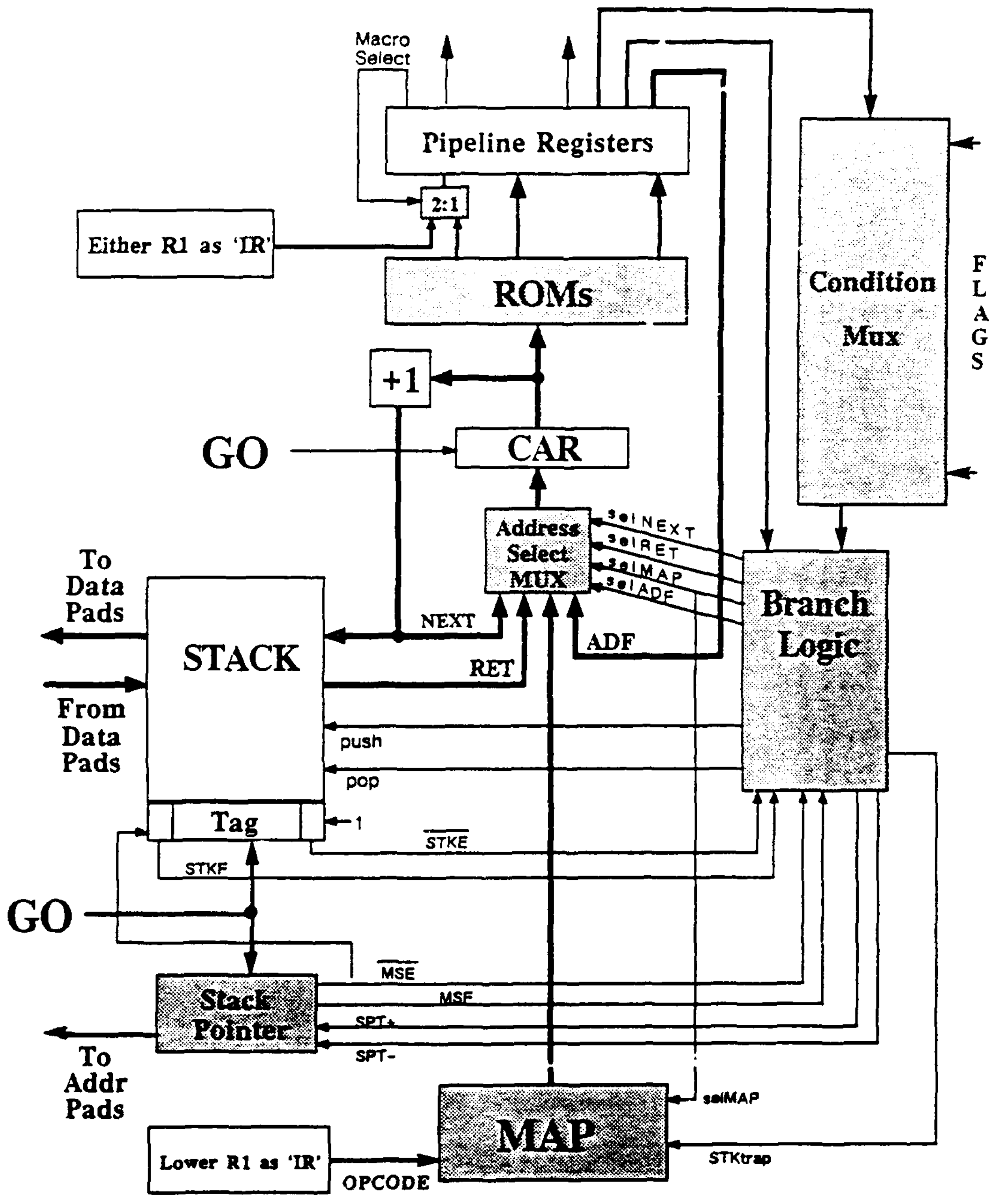


Figure 3.7. FPASP Microsequencer

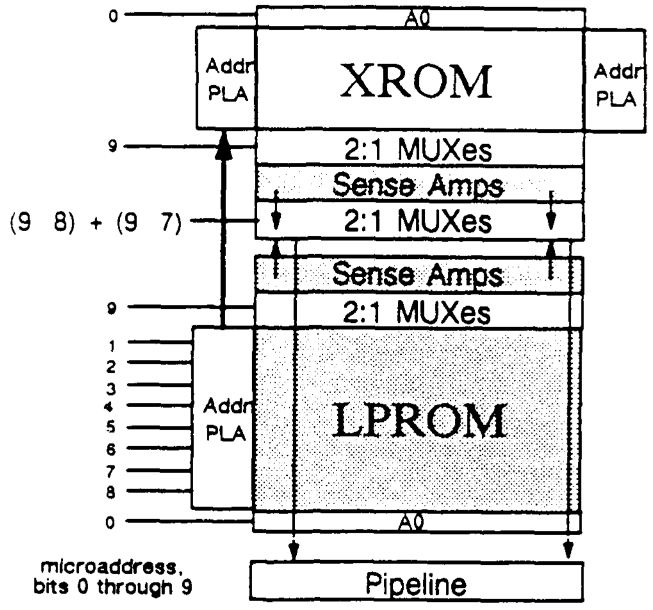


Figure 3.8. Microcode ROM Arrangement

Condition Multiplexer: The condition multiplexer collects all of the flags generated throughout the FPASP. The multiplexer is then used to select the particular flag that will be used for a branch condition. Six bits in a "Conditional Multiplexer Select" field of the upper ROM word are used to control the selection by the MUX. Of the 64 total selections provided by the field, there are 48 distinct flags and the remaining 16 are the inverse of other flags.

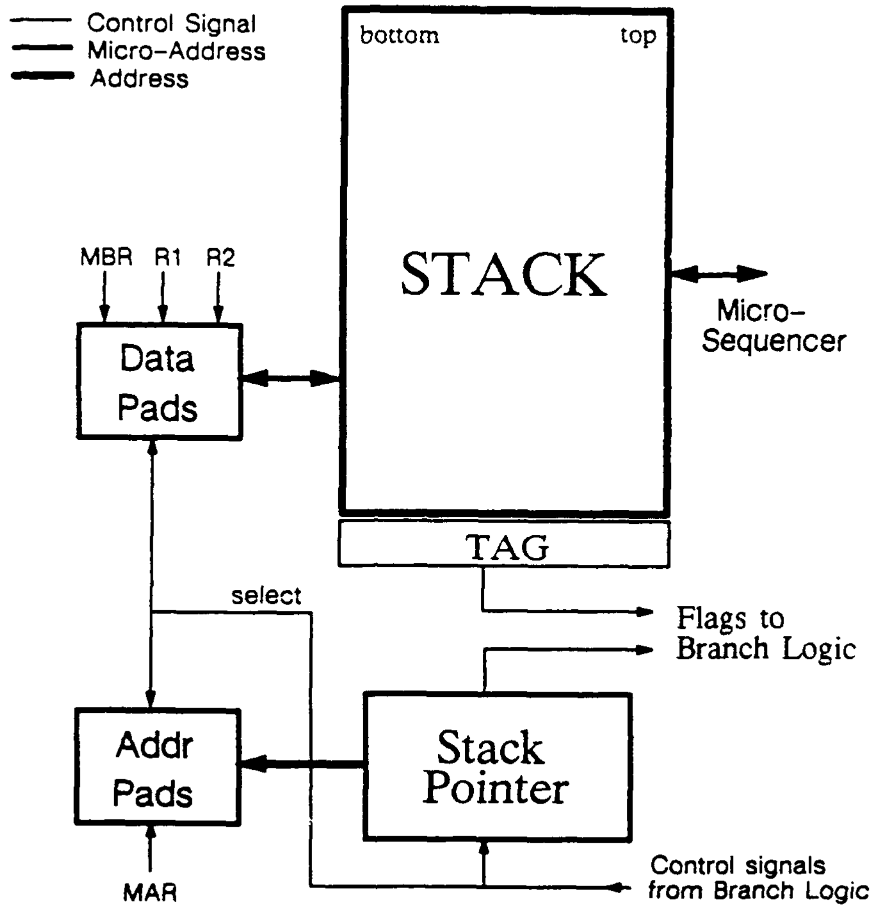


Figure 3.9. Stack Architecture

IV. FPASP RTL Simulator

4.1 Introduction

The design and development of the Kalman Filter microcode for the FPASP requires the ability to test and simulate the microcode not only at the submodule level, but also at the overall system level. The original requirements for an FPASP simulator required the simulator to be developed using VHDL. Besides allowing a user to simulate a microcoded user algorithm, VHDL also provides the means to validate a hardware design. After a considerable amount of work, it was determined that the time required to simulate a fairly large microcode program would be quite large. In fact, preliminary analysis indicated that a simulation a 64–state Kalman Filter using a VHDL simulator would require at least 2 years of real time on a VAX 8800. Even if the preliminary data was off by an order of magnitude, the time to simulate a large Kalman Filter would still be unacceptable. In its current form, VHDL does not serve as an effective platform for the development of application software.

The need to test microcode applications developed for the FPASP required that an application specific language be used to allow for rapid simulations. C was chosen as the language to develop the FPASP RTL simulator because it is extremely flexible, faster than most other higher–order languages, and easily ported to other computer systems. This chapter will document the development of the FPASP RTL simulator to include the requirements the simulator had to meet, the external interfaces of the simulator, a detailed functional description of the simulator, testing the simulator, limitations of the simulator, and some notes on the use of the simulator. A complete users manu . is provided with the simulator.

4.2 RTL Simulator Requirements

The requirements the RTL simulator had to meet were varied but quite basic. The RTL simulator had to be able to read in directly the *.trans* file output of the GMAT (Generic Microcode Assembler Tool) assembler. The GMAT assembler was developed by Robert Hauser to support the development of FPASP microcode, and to generate the input file required by the XROM silicon compiler for the layout of the fixed ROM storage (Hau87). Along with generating files compatible as inputs to the XROM compiler, the GMAT assembler also generates a file, with a file extension of

.trans, that contains a bit string translation of the microcode. Essentially, the bit string translation is the ASCII equivalent of the XROM/LPROM bit pattern contents. The requirement that the RTL simulator accept this *.trans* file as the XROM/LPROM input source allows the microcode designer to develop mnemonic microcode programs, assemble them with GMAT, and then execute them with the RTL simulator. This requirement greatly enhances the development of microcode for the FPASP by allowing the designer to work at the mnemonic level or essentially one level of abstraction above the actual bit patterns of the microword itself.

A second requirement of the RTL simulator was the need to provide sufficient output to aid the designer in debugging the microcode under development. Information passed to and from the FPASP must be accomplished through the FPASP's external memory. The information can be anything from the data the FPASP must perform some operation on to flags indicating some internal condition. This however quickly proved to be insufficient as a means to track the FPASP's internal flow of data required to trace a particularly complex program. To aid the designer, the RTL simulator had to be able to provide some insight into the internal flow of data as the simulation progresses. One means to accomplish this, and the one which was pursued, was to provide *snapshots* of the internal register states as well as the current data on the busses for each simulated clock cycle if so desired. This option is completely user selectable, and the user can elect to record just the bus information or a complete record of register contents and bus values. In addition to the types of information recorded, the user can also select at which address in the microcode program the recordings will begin. Using this approach, a designer will be able to review the progression of a program as it was executed step by step.

A third requirement of the RTL simulator was the ability to execute microcode programs as quickly as possible to provide timely simulation results. It is anticipated that the execution time of microcode programs, in particular the Kalman Filter program, can be quite large requiring on the order of a million simulation cycles for a complete simulation. To meet this requirement, the development of the RTL simulator had to pay close attention to the methods used to model the individual FPASP components to ensure that the correctness of the model was not sacrificed for speed of execution.

Finally, and just as important as modeling the internal functions, the RTL simulator had to model the internal components of the FPASP topologically. The FPASP is a very complicated example of VLSI architecture which is comprised of many individual components that must be sequenced correctly according to data and control signal inputs. It is not enough to develop a program that takes the same inputs and provides the same outputs as the FPASP, yet internally does not map to the FPASP, and call it a simulator. The simulator developed to simulate FPASP microcode must internally map to the individual sub–components of the FPASP. This requirement will insure that the simulator can be easily updated to accommodate future modifications to the FPASP architecture.

4.3 External Interfaces

The external interfaces to the FPASP RTL simulator are the disk files that must be present in the current directory when the simulator is executed. There are three input files required by the RTL simulator. One of these files has already been described and is the GMAT generated *.trans* file that contains the initial XROM/LPROM contents. The other two input files contain the simulated external memory contents - one for the even memory bank, and one for the odd memory bank.

The RTL simulator produces three output files at the completion of each simulation run. The first output file, MEMDUMP, contains the final contents of the simulator's external memory banks in two columns. The first column contains the even memory bank data, and the second column contains the odd memory bank data. A third column is also generated giving the floating point representation of the memory data. The second output file generated by the RTL simulator, FPASP.OUT, contains the sequenced recordings of the simulator's internal register states and bus values. There is one record for each simulated clock cycle and, depending on the size of the microcode program being simulated, the file can be quite large. The third output file, FPASP.RPT, simply contains the total number of clock cycles for the simulation, and the total number of floating point operations performed. This information is useful for evaluating the performance of a given microcode program.

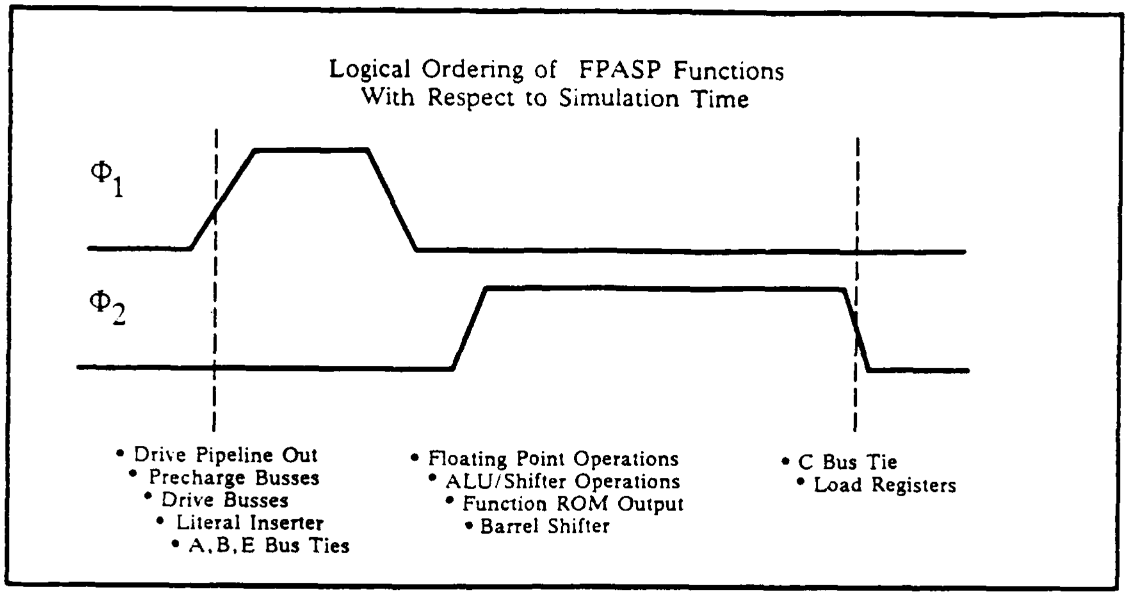


Figure 4.1. RTL Simulator Sequencing

4.4 RTL Simulator Development

The development of the RTL simulator began with the identification of each internal FPASP function based on the register level design given in Chapter 3. The operation of each function could then be modeled with a C subroutine. The sequencing of the internal FPASP functions is based on an externally generated clock signal. Simulating this timed sequencing of events required that each function, whether it be loading a register or driving a bus, be logically ordered according to when it should execute with respect to the system clock. Figure 4.1 lists the FPASP functions relative to the clock signal and consequently identifies the order of execution of the function subroutines. For functions that drive register contents onto a bus, or functions that load bus values into registers, this method is straight forward. For example, Figure 4.1 clearly indentifies that the **A** and **B** busses need to be driven by registers befo- ; ALU or floating point subroutines can be called, but what's not so clear is the sequencing of the bus ties.

To simulate the actions of the bus ties, it is necessary to insure that the undriven bus is being set equal to the driven bus, and not the other way around. In the case of the pre–charged buses which consist of the **A**, **B**, and **E** busses, this is accomplished by simply ANDing the upper data path and the lower data path busses together and setting both busses equal to the result. However, for the **C** bus which is not pre–charged, it is necessary to explicitly determine which bus has being driven and setting the undriven bus equal to the driven bus. In the case where both upper and lower **C** buses are driven during the same clock period, an error condition is flagged. After determining the order of events in a given simulation cycle and resolving the bus ties, the design and development of the RTL simulator proceeded by grouping similar functions that could occur during the same instant of simulation time, and writing subroutines to emulate their operations.

The development of the individual subroutines that emulate the various functions of the FPASP was based on the similarity of the functions to be simulated. For example, all FPASP functions that placed register values on the busses during the rising edge of Φ_1 were grouped into one subroutine for each data path. In a similar fashion, all FPASP functions that loaded bus values into registers with the falling edge of Φ_2 were placed in another subroutine. Calling the subroutines consecutively effectively simulates the MS flip–flop operation of the registers. There are 15 basic subroutines that are called in consecutive order to simulate the data flow through the FPASP. Figure 4.2 gives a flow chart depicting the 15 subroutines and the order in which they are called from the main routine.

Figure 4.2 shows that the simulation of a microcode program begins with the initialization of the simulator. After initialization, the first subroutine called is the microsequencer, after which the pre–charged busses are charged, and then register contents are driven onto the buses according the bit patterns of the individual micro–fields. Once the selected registers have driven their contents onto the busses, the **A**, **B**, and **E** bus tie micro–fields are tested, and the appropriate busses are tied together if so selected. Next, the Function ROM, Literal Inserter, and Barrel Shifter subroutines are called. These subroutines may have to place data on the busses to be used by the computational subroutines that follow. The process continues by performing selected upper ALU/SHIFTER functions followed by selected lower ALU/SHIFTER functions. Next, the appropriately selected floating point multiplication functions are performed. Since floating point multiplication is a default micro–field setting, floating point multiplication is performed every clock cycle unless integer multiplication is explicitly selected. Floating point addition is performed next unless floating point subtraction is explicitly requested. Each of the subroutines that process data place their results on the appropriate **C** bus, and the next two routines take the value off the **C** busses and load it into the selected registers. The exceptions are the floating point processing routines which pass their results into delay variables which effectively emulates a pipeline. This is to insure the proper two

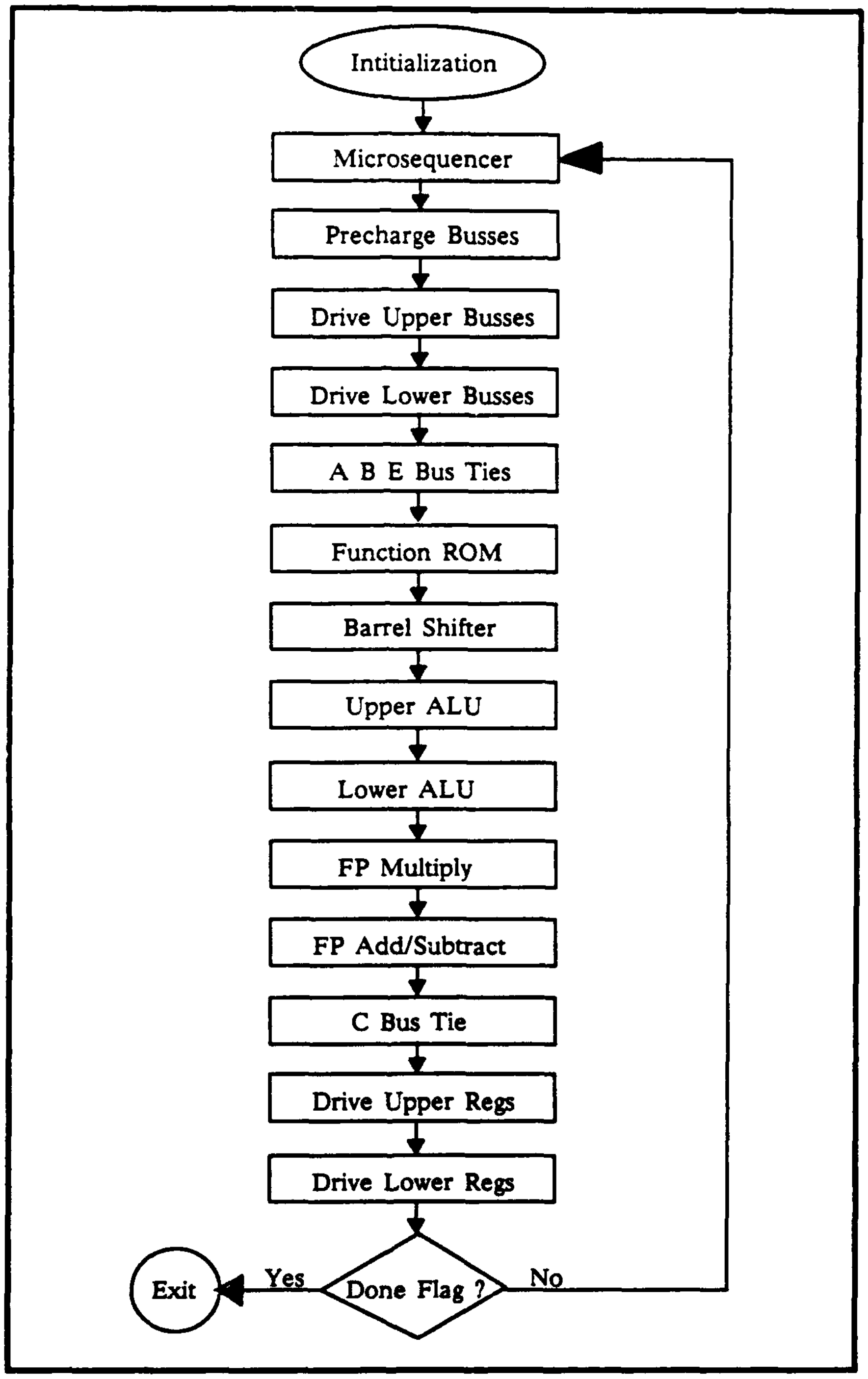


Figure 4.2. RTL Simulator Flow Chart

clock cycle delay required for floating point operations.

4.5 RTL Simulator Functional Description

The preceding section described the development process for the FPASP RTL simulator and briefly described the subroutines that were developed to emulate the various FPASP operations. This section will discuss in depth each of these routines to illustrate that the routines do in fact emulate the portions of the FPASP they were designed to.

4.5.1 Initialization The initialization routine reads in the microcode program from the *.trans* file into an array that serves as the FPASP XROM/LPROM. Currently, there is no distinction between the XROM and LPROM and all microwords are stored contiguously. The initialization routine also reads in the memory contents from the two external files, HI_MEM and LO_MEM, and places the data in two arrays that serve as the even memory bank and the odd memory bank.

4.5.2 Microsequencer: The microsequencer subroutine was designed to emulate the operation of the FPASP microsequencer to include the functions of the pipeline register, control address register (CAR), microaddress stack, and the XROM/LPROM. In addition to these functions, the microsequencer subroutine also provides the capabilities to multiplex all the various flags generated throughout the FPASP that control conditional branches, calls, and returns from within a microcode program.

Initialization of the simulator insures that the CAR is preset to zero to point to the first word of the micro ROM. The sequencing of events within the microsequencer subroutine begins with copying the first 64 bits of the pipeline register variable and placing those bits into the upper ROM word, and copying the last 64 bits from the pipeline register variable into the lower ROM word. The pipeline register variable is then updated with the next micro word from the micro ROM as indexed by the CAR. The Branch Control micro–field bits from the upper ROM word and are evaluated to see if a branch, call, return, or map was requested. If no branch is requested in the Branch Control micro–field, the CAR is simply incremented. If a branch is requested, the subroutine converts the Conditional Multiplexer Select micro–field bits from the upper ROM word into an integer, and indexes into a boolean flag array to test the requested flag. If the flag is found true, the Next Address/Literal Field micro-field from the lower ROM word is converted to an integer. The CAR is then updated with the new address and the subroutine exited. If the flag is false, the CAR is incremented and the routine exited.

If a call to a microcode subroutine is requested, the flag pointed to by the Conditional Multiplexer Select micro-field is tested and if true, the current address in the CAR is incremented, pushed onto the micro stack, and the stack pointer incremented. The address in the Next Address/Literal Field micro-field is fetched, converted to an integer, and placed in the CAR. Here again, if the flag tested was false, the CAR is incremented and the routine exited.

In a similar fashion, if a return from subroutine was requested, the appropriate flag is tested and if found to be true, the stack pointer is decremented and the return address is popped from the micro stack and assigned to the CAR. Otherwise the CAR is incremented and the routine exited. Currently, the unconditional MAP Branch Control is not implemented. The MAP Branch Control functions to support assembly language routines written for the FPASP which at this time are not clearly defined.

4.5.3 Bus Precharging: Prior to driving any data from the registers onto the **A**, **B**, or **E** busses in both data paths, the busses must be precharged. The precharging of the busses is performed in the top level routine and consists of writing 1's into every bit of the busses.

4.5.4 Drive Upper Busses: The subroutine Drive_Upper_Busses functionally groups all FPASP operations that drive register contents onto the **A**, **B**, or **E** busses. This was necessary to prevent the proliferation of many small routines that needed to occur at essentially the same time in the simulation cycle. This subroutine first examines the Upper A Bus Select micro-field of the upper ROM word and selects the appropriate register from the general purpose registers, the incrementable registers, the memory pointer registers, or the upper MBR. The upper **A** bus, which is precharged, is AND'd with the contents of the selected register. Similarly, the Upper B Bus Select micro-field of the upper ROM word is scanned and contents of a selected register, or the upper data path outputs of the floating point multiplier and floating point adder/subtractor, is AND'd with the precharged upper **B** bus.

The functions of the literal inserter are also emulated in this subroutine. There are 15 functions the literal inserter can perform based on the bit pattern of the Literal Inserter Control micro–field of the upper ROM word. The first three functions which are specific to the assembly language capabilities of the FPASP are not supported in this simulator. The other 12 however are completely implemented. The Literal Inserter Control micro–field is scanned and if a request to insert the bit string located in the Next Address/Literal Field micro–field of the lower ROM word is found, the 16 bits of the Next Address/Literal Field micro–field from the lower ROM word are AND'd onto the **A** busses according to the criteria defined in the FPASP Microcode Fields Definitions in Appendix A.

This Drive_Upper_Busses subroutine also drives the upper **E** bus with the contents of the two memory pointers. The Upper E Bus Select micro–field is scanned and based on the bit pattern encountered, either the contents of the APTR or the BPTR are placed onto the upper **E** bus. This micro–field also controls disconnecting the upper memory address and controls allowing them to float to high Z. This function is related to the assembly language capabilities of the FPASP and is not implemented in this simulator.

The upper MAR is directly connected to the upper Address bus of the FPASP and this subroutine ensures the upper Address bus is driven with the current contents of the upper MAR. During each simulated clock cycle, the contents of the upper MAR are driven onto the upper Address bus.

The final function the subroutine drives of the upper **D** bus. Data can be driven onto the upper **D** bus from either the upper MBR or from the Upper Memory Bank. The subroutine checks the Upper Memory Chip Select Bar micro–field and if the bit is '0' indicating the chip is active, the routine procedes to check the Upper Memory Write Enable Bar micro–field and the Upper Memory Chip Output Enable Bar micro–field. If the Upper Memory Write Enable Bar micro–field is '0', and the Upper Memory Chip Output Enable Bar micro–field is '1', the external memory chips are not enabled for a read operation and the contents of the upper MBR are driven onto the upper **D** bus. If the Upper Memory Write Enable Bar micro–field is '1', and the Upper Memory Chip Enable Bar micro–field is '0', the external memory chips are enabled for a read operation and the upper **D** bus is driven from the Upper Memory Bank.

4.5.5 Drive Lower Busses: The next subroutine called is the Drive_Lower_Busses routine is a mirror image of the Drive_Upper_Busses subroutine with the following exception. The function of the literal inserter is not present in this routine. The literal inserter does have the capability to insert literal fields from the microcode directly onto both **A** busses. Rather than break the operation of the literal inserter into two separate routines, one driving the upper **A** bus and the other driving the lower **A** bus, it was more efficient to leave the routine intact and place it in the Drive_Upper_Busses routine.

4.5.6 Barrel Shifter: This subroutine emulates the function of the Barrel Shifter. The Barrel Shifter Set–up micro–field is sliced from the lower ROM word, as is the Barrel Shifter Control micro–field, and both are converted to integers. If the Barrel Shifter Set–up micro–field is zero and the Barrel Shifter Control micro–field is not zero, the data on the lower **A** bus is circularly shift left the appropriate number of times. The result is placed on the lower **C** bus. In a similar fashion, the rest of the Barrel shifter Set–up micro–field is decoded, and the appropriate shifts made. In those cases where the MSB of the Barrel Shifter Set–up micro–field is '1', the Barrel Register is loaded with the five LSBs from the lower **C** bus before the routine terminates. The value loaded into the Barrel Register will be available in the next simulation cycle.

4.5.7 Function ROM: This subroutine emulates the operation of the Function ROM. The Function ROM Select micro–field is sliced from the upper ROM word and converted to an integer. If the result is not zero, it is used as an index to select one of the Function ROM pages. The Function Rom itself is maintained as a two dimensional array of string constants that contain the seed values for various recursive routines. After selecting the proper page based on the integer value of the bit string of the Function ROM Select micro–field, bits 52 downto 47 are masked from the upper **B** bus and shifted right 16 times. These bits represent the LSB of the exponent and the 4 MSBs of the mantissa of the most significant 32 bits of a floating point number. The resultant bit string is converted to an integer and used to select the appropriate seed from the selected ROM page and is driven onto the upper **C** bus.

4.5.8 A, B, and E Bus Ties: At this phase of the simulation cycle, all of the busses, both lower and upper, that are going to be driven have been updated with new values. This routine examines the A Bus Tie micro–field and the B Bus Tie micro–field of the lower ROM word and the E Bus Tie micro–field of the upper ROM word. As noted earlier, the **A** busses, **B** busses, and the **E** busses are precharged busses. This routine examines the A Bus Tie, B Bus Tie, and E Bus Tie micro–fields, and if any of the bits are a '1', the appropriate upper and lower busses are AND'd together and the resultant value reassigned to both the busses.

4.5.9 Upper ALU/SHIFTER Functions: This routine combines the functions of the upper ALU and the upper Linear Shifter. At this phase of the simulation cycle, both the upper **A** bus and the upper **B** bus are driven with integer values current, and the data can now be processed. The routine emulates all the functions of the ALU collecting the results in an intermediate shifter bus. The routine then takes the results passed from the ALU on the shifter bus and emulates all the functions of the Linear Shifter. The results of the Linear Shifter are passed out on the upper **C** bus. During the simulations of the ALU and the Linear Shifter, any flags that change as a result of the selected functions are set accordingly.

The routine begins by examining the Upper ALU Select micro–field of the upper ROM word. The bus width of all the busses, except for the upper and lower address busses, are 32 bits which matches the number of bits in the long integer type in the C programming language. This routine treats the **A** and **B** bus inputs as two long integers and performs the logical and arithmetic functions using the C's internal integer routines. This method posed the problem of setting the flags correctly as the C programming language does not provide access to its internal flags. The subroutine works around this by performing a few bit manipulations on the input values, and by testing the output value. To test for a zero condition, the output value is simply tested for zero. Testing the MSB of the output value determines if the value is negative. To test for a carry condition, the most significant 31 bits of both input values are masked off. The modified input values are then added and the sum shifted right once leaving only the carry out of the LSB position. The original input values are then shifted right once and added. The carry out of the LSB position is then added into the sum and the MSB position of the final sum tested for a carry out condition. To test for an overflow condition, the MSB of the original input values is masked off, and the modified values added. The MSB of the sum is compared against the previously determined carry out to determine is an overflow has occured.

The output of the ALU is placed on an intermediate shifter variable which acts as the input bus to the Linear Shifter. The routine then examines the Upper Shift Control micro-field to determine which shift operation. if any, is to be performed. Here again the input on the shifter bus is treated as a long integer and C's internal logic routines are used to perform the various shifts. Before a shift is performed, the MSB or the LSB of the input is tested, depending on the direction of the shift, to determine if the upper Shift Out flag should be set.

4.5.10 Lower ALU/SHIFTER Functions: This routine combines the functions of the lower ALU and the lower Linear Shifter. The functional description for the routine is identical to that given for the upper ALU and Linear Shifter and therefore will not be reneated.

4.5.11 Floating Point Multiplier: This routine emulates the functions of the floating point multiplier by combining the halved floating point numbers on the **A** and **B** busses, performing double precision multiplication, splitting the product in half before placing it on the **C** busses, and setting the appropriate flags. The routine combines and splits the floating point numbers by making use of the C union structure. The union structure allows different data types to occupy the same physical memory space. In this routine, a union structure is defined to hold a double precision floating point variable as well as an array of two long integers. Since a double precision floating point variable occupies eight bytes and two long integers also occupy eight bytes, the two completely overlap one another. Addressing the union as a floating point variable allows floating point operations to be performed, yet addressing the elements of the double integer array allows integer operations to be performed on each element. To combine the two halves of a floating point number from the **A** busses merely requires assigning the contents of one bus to an element of the double integer array, and the contents of other bus to the other element of the array. In a similar fashion, the data on the two **B** busses can be combined within a similar union structure. Double precision floating point multiplication can then be performed by multiplying the two union floating point variables together and placing the product in yet a third union's floating point variable. The resultant union's integer array can then be indexed into to retrieve the upper and lower halves of the floating point product from the two integer array elements which can then be placed on the **C** busses. Although this method provides a very fast an efficient means to emulate the FPASP floating point hardware, it does provide difficulty in determining which flags need to be set. To determine if the result was a zero only requires checking the result placed on both the **C** busses to see if they are both zero. To determine if the Not–A–Number flag should be raised required checking the resultant exponent for all ones. To determine if the underflow, overflow, and denormalized flags should be raised would require extensive bit manipulations of both the multiplier and the multiplicand. This along with the default floating point multiplication that occurs every simulation cycle would significantly increase the simulation times. Therefore, these flags are not supported at this time in the RTL simulator.

The results of the floating point multiplication routine are pipelined to provide a simulated two cycle delay. Although pipelining the results is sufficient to provide the required delay, it is possible to input new data into the emulated floating point multiplier every clock cycle and receive correct results with a two cycle latency.

The second function this routine emulates from the floating point hardware is integer multiplication. As explained earlier, the contents of the **A** and **B** busses are placed in double integer arrays in union structures. This allows for direct integer multiplication between the elements of the two arrays which is the method chosen to emulate the FPASP integer multiplication. Although it is an easy matter to determine if an overflow occurred during the integer multiplication, it would be very difficult to determine how much of an overflow occurred. For this reason, integer multiplication where the product exceeds 32 bits is not supported by this emulation method although the overflow flag is set to indicate the result is erroneous. As with the floating point multiplier, the results are pipelined to provide a two clock cycle latency and care should be taken in the developing of microcode programs to insure the multiplier is not being loaded in a pipeline fashion.

4.5.12 Floating Point Addition/Subtraction: Emulation of the floating point adder/subtractor uses the same union structure approach as described for the floating point multiplication emulation. Complete double precision floating point addition and subtraction is supported, but as with floating point multiplication, several of the flags are not supported. The Adder Overflow flag and the Adder Difference flag are not supported because of the number of bit manipulations that would have to be performed in a default function that is performed every simulation cycle. This however should not be detrimental to the successful development of microcode programs. The results of the floating point adder/subtractor are pipelined to simulate the two cycle latency of the hardware.

4.5.13 C Bus Tie: At this phase of the simulation cycle, the subroutines that emulate the processing functions of the FPASP are complete with their corresponding results placed on the appropriate **C** bus. As discussed earlier, the **C** bus is not a precharged bus and consequently the upper and lower **C** busses can not be ANDed together should the C Bus Tie micro-field request it. The reason for this is that old data doesn't dissipate from the busses as it would if the busses were precharged and ANDing the busses together would produce erroneous results. The RTL simulator solved this problem by providing two global flags, one for each **C** bus, that are set by the individual processing routines whenever they drive results out onto the bus. Initially, at the beginning of each simulation cycle, these flags are reset to false. After the processing routines have completed, the C Bus Select micro-field is scanned and if the bit is found set to '1', this routine checks the bus flags to determine which **C** bus was driven. The **C** bus that wasn't driven during the current simulation cycle is set to equal the driven bus, or if both busses were driven, the contents of both busses are ANDed together and both busses are set equal to the result.

4.5.14 Drive Upper Registers: During the final phase of the simulation cycle, the processed results which were driven onto the **C** busses are latched (or assigned) into registers indicated by the various micro-fields of both ROM words. Additionally, the incrementable registers, which include the incrementable registers, memory pointers, and the MARs, are updated at this time.

The subroutine starts by converting the Upper C Bus Select micro-field from the upper ROM word bit string into an integer. If the integer is in the range from 1 to 25, the corresponding general purpose register is loaded with the contents of the upper **C** bus. If the integer is greater than 25, the appropriate incrementable register, memory pointer register, or the MBR is loaded with the contents of the upper **C** bus. Next the Upper Memory Address Register Control micro-field is scanned and the MAR is updated with the contents of the upper **C** bus, upper **E** bus, or the register is incremented according to the micro-field bit pattern. The Upper Memory Buffer Register Control micro-field is scanned next and either the general purpose register 1, or the general purpose register 2, or the MBR is loaded with the contents of the upper **D** bus. The subroutine next scans the Upper Incrementable Registers Control micro-field and the selected incrementable register is incremented. This is followed by a scan of the A, B Pointer Controls micro-field. In this portion of the subroutine, the memory pointer increment registers are loaded from the **C** bus, or the memory pointer registers are incremented with the value of their increment register, or a combination of the two is performed. Finally, if a write to external memory is requested according to the bit settings of the Upper Memory Chip Select Bar, Upper Memory Write Enable Bar, and Upper Memory Chip Output Enable Bar micro-fields, the contents on the upper **D** bus are written to the external memory location pointed to by the upper address bus.

4.5.15 Drive Lower Registers: The subroutine performs the identical functions as its counterpart described above. The exception is that all the functions are performed on the lower data path registers and the lower external memory bank.

4.6 RTL Simulator Testing

The RTL simulator test plan required that several small microcode programs be developed to exercise the various data processing capabilities of the simulator. At the end of each test simulation, the results which were recorded in the simulation *snapshot* file, were compared against hand generated anticipated results and the iterative process of testing and debugging continued until the two results matched.

A microcode test program was written to test the transfer of data between general purpose registers. This program effectively tested driving data out from a register onto the **A** bus, passing the data through the ALUs and the Linear Shifters to the **C** bus, and back into a selected register. Next a microcode test program was developed to test the functions of the ALUs. This program drove the data out from two registers onto the **A** and **B** busses and into the ALUs. The 15 functions of the ALUs were exercised one at a time and the results placed into separate registers where they could be compared to the anticipated results. Similarly, a microcode test program was developed to test the 15 functions of the Linear Shifters and the results were also compared against anticipated results. After testing the ALUs and the Linear Shifters, microcode programs were written to test the functionality of the floating point multiplier and the floating point adder/subtractor. Next, programs were written to test the literal inserter, bus tie functions, and external memory. Once testing of the individual subcomponents was completed satisfactorily, more elaborate microcode programs were used that would integrate all the functions of the simulator.

Capt John Comtois, as part of his thesis, developed microcode routines for matrix multiplication and for Newton Raphson Inversion. These routines were assembled using GMAT and simulated with the FPASP RTL simulator as the final phase of the test plan. Both the matrix multiplication routine and the Newton Raphson routine have many conditional branches and subroutine calls which thoroughly tested the microsequencer emulator. The matrix multiplication routine performs a number of floating point multiplications and additions based on the dimensions of the two matrices being multiplied. Additionally, the matrix multiplication routine exercises the incrementable registers and memory pointer registers along with numerous reads and writes to external memory. The Newton Raphson Inversion routine exercises the Function ROM, the floating point emulators, certain functions of the ALUs, the literal inserter, and relies on the ANDing function of the busses where more than one register drives the bus at one time. Testing the simulator using these routines required several iterations to correct the simulator and the routines themselves. After the simulator was capable of executing the microcode routines successfully using a variety of initial variables, the FPASP RTL simulator was ready to aid in the development of the Kalman Filter microcode program.

4.7 *FPASP RTL simulator Limitations*

A considerable amount of effort was expended to make the FPASP RTL simulator emulate as many of the functions of the FPASP as possible. As previously mentioned, the outputs of the floating point emulators are pipelined to emulate a two cycle latency. This however does not correctly emulate the actual FPASP floating point hardware. The two cycle latency of the FPASP floating point hardware is a result of signal propagation delays and not because of any pipelining techniques. Results read before two clock cycles will be completely erroneous and may not even be floating point numbers. The methods used to emulate the floating point hardware in the RTL simulator provide no easy methods of determining the conditions of all the flags. Consequently, the multiplier overflow, underflow, and denormalized flags are not supported, and the adder overflow and difference flags are not supported. This may be changed at a later date. However, the current capabilities of the simulator are more than adequate to support the development of the Kalman Filter microcode program.

4.8 Using the FPASP RTL simulator

Complete instructions for using the FPASP RTL simulator to run FPASP microcode programs are given in the simulator's users manual. However, it appears appropriate to say a few words on using the simulator at this time.

Three files must be prepared before the RTL simulator can be used. One of the files contains the ASCII bit string XROM/LPROM contents, and was generated using GMAT to assemble the mnemonic microcode program. This file will have a *.trans* file extension on UNIX machines and a *.trn* file extension on IBM PC machines. The next two files are named HI_MEM and LO_MEM and contain the external memory contents that will be read into the emulated upper external memory bank and the lower external memory bank respectively. All floating point numbers have been split in half, and the most significant half is in the HI_MEM file and the least significant half is in the LO_MEM file. The user will have to provide a program that will perform this operation on the data to be used by the simulator. Once these files are generated and located in the same directory with the FPASP RTL simulator, the user has two optional means to invoke the simulator. At the system prompt, the user can enter FPASP followed by the simulator options and press enter as shown below.

```
%   FPASP   filename.trans   y/n   XXX   y/n
```

filename.trans	microcode program
y/n	generate FPASP state file
XXX	line number to start recording
y/n	abbreviated state file

This option allows the user to invoke the simulator specifying all the options from the command line. This is particularly useful by allowing the simulator to run in the back ground on UNIX based computers. With the second option, the user enters FPASP at the system prompt, and is queried with a series of prompts asking for the same information described above.

When the simulation is complete, the simulator generates three files. The first file, named FPASP.OUT, contains the FPASP state information collected during each simulation cycle beginning at the specified line number of the microcode program. The second file, named MEMDUMP, contains the contents of both the upper and lower external memory banks in two columns with each column entry numbered indicating the memory index. A third column gives the floating point representation of the data contained in the two memory columns. The third output file contains performance information indicating the total number of clock cycles required for the program, and the total number of floating point operations performed.

V. VHDL Simulator

5.1 Introduction

The previous chapter gave a detailed account of the development of the FPASP logic simulator using the C programming language. The logic simulator emulates the functional elements of the FPASP using subroutines and procedure calls sequenced in the correct order. The usefulness of this type of simulator is limited to emulating the functional concepts of a digital system, and is not capable of testing and validating the VLSI design itself. VHDL, on the other hand, is a formal notation intended to express the functional and logical organization of a digital system. The function of a system is specified as a transform mapping of inputs to outputs where time is explicitly referenced in the transform, and the organization of a system is expressed in terms of interconnected components (Int85). In this manner, VHDL provides the necessary tools to completely describe a digital system in a model that is executable, and is therefore capable of aiding in the testing and validation of a VLSI design. The concept of VHDL departs so radically from the more traditional higher order languages that it would benefit to provide a brief description of VHDL.

5.2 What is VHDL

VHDL has been compared to ADA and in fact has many of the same goals as the ADA language. VHDL provides a wide range of abstraction levels from the architecture level down to the gate level (Coe89). More importantly, VHDL supports the easy mixing of these widely varying levels of abstraction during simulation making it possible to adopt a true top-down design style. In addition, the VHDL language has exceptional facilities for modeling hardware timing.

In VHDL, signals are objects that may be changed and have a time dimension (Arm89). Associated with every signal assignment statement is a signal driver which has a value component and a time component. For a given transaction, the value component represents a value that the signal will assume at some point in time, and the time component specifies which point in time. If the name of a user-defined resolution function appears in the declaration of a signal, or in the declaration of the subtype used to declare the signal, the signal is a resolved signal (Fle88). If a signal is to have more than one driver, the signal must be a resolved signal. The user-defined resolution function is evaluated when one or more of the drivers of the signal receives a new value.

The resultant value assigned to the signal is the value computed by the resolution function based on the inputs of all the drivers. If it is desired that the values of one or more drivers not be considered by the resolution function in computing a new signal value, those drivers must be turned off. Only the drivers of guarded signals can be turned off.

If a signal is of the kind **register** or **bus** then the signal is a *guarded* signal. A guarded signal is assigned values under the control of Boolean-valued guard expressions. When a guard becomes False, the drivers of the corresponding guarded signals are assigned a null transaction to cause those drivers to turn off.

VHDL supports three distinct styles for the description of hardware architectures. The first of these is the *structural description*, in which the architecture is expressed as a hierarchical arrangement of interconnected components. The second is *data-flow description*, in which the architecture is broken down into a set of concurrent register assignments, each of which may be under the control of gating signals. The last is *behavioral description*, in which the transform mapping of inputs to outputs is described in sequential statements that look like any modern, high level computer programming language (Int85). All three styles may be intermixed in a single architecture.

The *design* entity is the primary hardware abstraction in VHDL. It represents a portion of a hardware design - cell, chip, board, or subsystem - that has well defined inputs and outputs, and performs a well defined function. Each design entity is described in two parts: the definition of the interface between the entity and the outside world, called the entity declaration; and a design for the input/output transformation, called the architecture body.

VHDL provides two modeling elements - the block and the process. A block statement defines an internal block representing a portion of a design, and blocks may be hierarchically nested to support design decomposition. Each block has a declarative section, an executable section, and may have a guard condition. When the guard condition is TRUE, concurrent statements which have been identified as guarded are enabled within the executable section. All statements within the executable section of a block are concurrent statements. A process statement defines an independent sequential process representing the behavior of some portion of the design. Each process has a declarative section followed by an executable section and may include a sensitivity list of signals that the process is sensitive to. If a transaction occurs in any of the signals present in the sensitivity list of a process, the process becomes active, and the statements in the executable section are executed in a sequential fashion. A process statement executes only once during a simulation cycle. VHDL allows blocks and processes to be intermixed within the architecture body of a design entity.

VHDL components are built up of one or more design entities, where block statements and process statements are used within the architecture bodies of the entities to model the structure and behavior of the components. The individual components that comprise the functional sub-units of a digital design are instantiated in a top-level design unit to form the completed model of the digital system. This completed model can then be executed simulating the functions and operations of the digital system modeled.

5.3 VHDL Simulator Requirements

Like the logic simulator in the preceding chapter, the VHDL simulator must be able to emulate the functions of the FPASP. However, the means of emulation are significantly different. The logic simulator was designed to allow for quick and efficient simulation of microcode programs. To do so the logic simulator was developed using the C programming language to emulate the logic functions of the FPASP. This method, while it does provide for a very fast simulator, prevented complete implementation of all the floating point hardware flags, and is of little use in validating the hardware design of the FPASP. The VHDL simulator, while not providing the means to quickly simulate microcode programs, must be able to emulate the functioning hardware of the FPASP as a means to aid in the validation of the FPASP design.

The previous section indicated that VHDL provides for a broad range of abstraction levels that can be used to model a digital system. The use of any particular level is dependent on the maturity of the digital design that is to be modeled. As stated in Chapter 2, the final design of the FPASP is proceding in a concurrent thesis based on the finalized register level description of the FPASP given in Chapter 3. This necessarily implies that design at the gate and circuit levels are still dynamic and subject to change as the finalized design nears completion. Therefore, the development of a VHDL hardware model of the FPASP will be done at the register level, and modeling of the individual sub-component behaviors will be done using procedure and function subroutines (Bor88). Using the register level of abstraction will require that the FPASP be decomposed into its basic sub-components at the register level, and a design entity developed for each subcomponent. This relegates the additional requirement that the behavioral architecture of each design entity be replaceable by a structural architecture at a later date when the design of the sub-component is finalized.

The VHDL simulator must be flexible. The flexibility of the VHDL simulator, like that of the logic simulator, is dependent on the ability to use a variety of microcode routines. This flexibility will require that the VHDL simulator is capable of reading in the contents of the GMAT generated *.trans* file into the XROM/LPROM. This requirement not only allows for any number of microcode routines to be executed using the simulator, but also for microcode development to be carried out at the mnemonic level. This requirement will directly aid the validation process of the hardware by allowing specific microcode routines to be executed which will only exercise the FPASP components under examination.

The VHDL support environment provides a Report Generator that is used after a simulation to generate a report which gives the values that were assigned to signals during the simulation run (Fle88). For simulations of small systems with none or very small bus widths, this tool is quite adequate. However, for models of large systems, the data presentation quickly becomes cumbersome and confusing indicating the requirement for an alternative approach for representing the data of the simulation run. In a fashion similar to that used by the logic simulator, the VHDL simulator must provide the means to record the internal register contents as well as the current data on the busses during each simulated clock cycle. Also the VHDL simulator must be able to write the contents of the simulated external memory banks out to external disk files for examination after the simulation run.

5.4 VHDL Simulator Development

The development of the VHDL simulator began with the identification of the individual FPASP components that needed to be modeled as design entities. Figure 5.1 depicts the finalized FPASP floorplan as it was presented in Capt John Comtois thesis. Each rectangular region represents one sub-component or macro-cell of the FPASP architecture and, taken together, provides the overall topology for the VHDL system model.

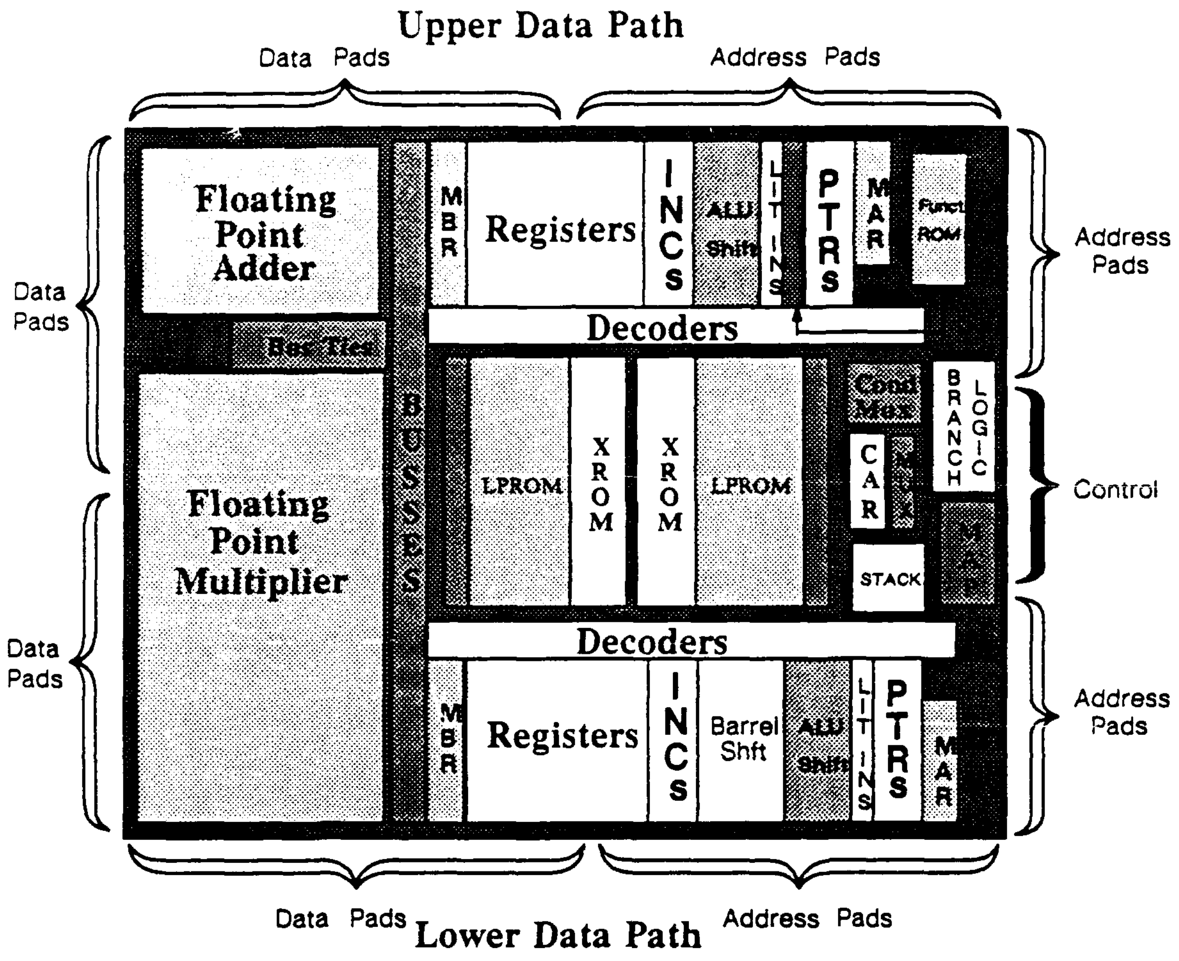


Figure 5.1. Final FPASP Floorplan

Decomposing the floorplan into its individual components provided the impetus for the design of the VHDL simulator, however, some of the individual components depicted in Figure 5.1 were combined with other components into single design entities. In particular, the MBRs, general purpose registers, memory address pointers, memory address registers, and the incrementable registers were combined, and the entire control section consisting of the CAR, MUX, Cond MUX, Stack, Branch Logic, and XROM/LPROM were combined. The registers were combined because from the register level, these registers are identical with the exception that some are incrementable. The combination of the control section was done because of the inter-dependence of each component. Aside from these two cases, the floating point adder, floating point multiplier, function ROM, upper and lower ALU/Linear Shifters, Barrel Shifter, and the Literal Inserter were identified to be modeled with design entities.

After the identification of the individual design entity requirements, the port mapping of each entity declaration was determined. Within each entity declaration, signals that import or export data to the entity architecture are identified in a port list. The level of the signals identified in the port list is tied directly to the level of abstraction used in the development the design entities. As previously noted, the FPASP VHDL simulator is designed at the register level of abstraction, therefore, signals in the port list are necessarily at that level also. For example, data input to a component comes from the **C** bus and data is outputed onto the **A** bus. The control for the component is derived by decoding its individual micro control field, and its operation is timed by the external clock. The signals that would be identified in the port list of the entity declaration for this component would be the **A** bus, the **C** bus, the micro control field slice of the ROM control word or the ROM control word itself, and the two phase external clock. Based on this example, and the fact that correct operation of the MS flip-flop requires a two phase clock input, the port list for each design entity was derived from the FPASP register level description shown in Figure 3.1.

The intra-module and inter-module timing requirements of the VHDL simulator were considered next. For the logic simulator, it sufficed to merely identify which FPASP functions had to occur before others, and which functions could occur at essentially the same time. The subroutines emulating these functions could then be called in a correct sequence from a top level routine. For the VHDL simulator, this approach would be totally inadequate for a register level system model. The sequencing of internal FPASP events are triggered by the rising and falling edges of a two phase external clock. Modeling this activity with VHDL is a straight forward matter. The two phase clock signals are mapped into the ports of each of the component design entities. These signals are now visible to the block statements and process statements that make up the architecture of the entity. Each block statement has an associated guard condition, and when the guard condition becomes "TRUE", the concurrent statements in the block are executed. Each process statement has an associated sensitivity list. The sequential statements in a process are invoked when a change occurs in one of the signals listed in the sensitivity list. To support invoking a block or process statement at the correct instant in simulated time, VHDL supports a rich repertoire of signal attributes. Invoking a block statement on the falling edge of a signal, the block's guard condition would include the boolean expression - (*signal* = '0' and not *signal*'Stable). The concurrent statements within the block would be executed with the falling edge of *signal*. The block would then be disabled until the next falling edge of *signal*. Invoking a process statement with the falling edge of a signal is slightly different. The signal can not be listed in the sensitivity list of the process because that would make the process sensitive to any change occurring in the signal. The alternate method of using the **wait until ...** statement causes the execution of the process to cease until the conditions it is waiting on are met. Placing the statement, **wait until** *signal* **= '0' and not** *signal*'**Stable** will cause the process to wait until the falling edge of *signal* before execution of the remaining statements resumes. Using these contructs, the execution of the design entities developed for the FPASP components can be timed in accordance to the rising and falling edges of the external clock.

The development of the FPASP VHDL simulator consisted of decomposing the FPASP architecture into its components (or macro- cells) at the register level. Each component was modeled in a design entity. The development of each design entity required determining the input and output signals to be identified in the entity declaration port list that were commensurate with a register level description. Finally, the intra-module and inter-module timing requirements were developed based on an external clock. The next section will detail the development of the individual entity architectures which model the behavior of the FPASP components. Included will be port list for each entity, the timing requirements the entity had to meet, and a description of how the component's behavior was modeled.

5.5 VHDL Simulator Functional Description

This section will describe in detail the individual design entities developed for the VHDL simulator. Design entities were developed for the VHDL simulator that modeled: the combination of the general purpose registers, MBRs, MARs, memory address pointers and their increment registers, and the incrementable registers; the FPASP control section consisting of the CAR, MUX, condition MUX, stack, branch logic, and XROM/LPROM; the ALU/Linear Shifter combination; the literal inserter; the bus ties; the function ROM; the barrel shifter; the floating point adder; and the floating point multiplier. In addition to these design entities, two other design entities are needed for the VHDL simulator. A design entity was needed to model the external memory that the FPASP must interface to, and a top-level design entity was needed to tie all the other design entities together. Before beginning with the detailed descriptions of the design entities, an important aspect of how VHDL handles signals needs to be clarified.

The VHDL simulator has the outputs of many registers tied to the same busses. The busses are represented as signals, and each register tied to the bus has a signal driver associated with it. The FPASP is designed so that more than one register can drive a bus at one time. The resultant value on the bus is the value obtained by ANDing the contents of the registers driving the bus. This therefore implies that the busses need to be resolved signals, and the resolution function needs to AND the values of the drivers of those registers selected to drive the busses. Those registers not selected to drive the busses need to have their drivers turned off implying the busses need to be guarded signals, and signal assignments to the busses need to be guarded assignments. It is clear that if the signal assignments to the busses are not guarded assignments, then all the registers connected to the bus will be contributing to the resolution function and the resultant computation of a new bus value. The preceding rational is also valid for registers that can receive new values from more than one bus.

As stated, the busses in the VHDL simulator are represented as signals. In the VHDL simulator, all busses are declared of the subtype BUS_TYPE. The subtype BUS_TYPE is 32 bits wide composed of the resolved signal ALU_BUS_BIT. ALU_BUS_BIT is an enumerated type that can assume the values of 'Z', '0', '1', or 'E'. If the value of a bus is all Zs, the bus is in a high impedance state indicating that no signal drivers are driving the bus. The values of '0' and '1' are the values of interest on the bus, and the final value 'E' indicates an error condition. The error condition is set by processing functions when an attempt is made to use the high impedance value of 'Z' in a computation. The resolution function performs a wired–AND function in that the final computed value is the result of ANDing all the signal driver values driving a signal.

It was explained earlier that block statements facilitate guard expressions. When the guard expression is True, all guarded signal assignments within the block are activated until the guard expression becomes False at which time the drivers associated with the guarded signal assignments are turned off. All assignments within a block are concurrent assignments, and if a particular behavior can be modeled with concurrent statements, this arrangement works well. If the behavior can not be modeled with concurrent statements, there are two alternatives. A function or a procedure which uses sequential statements can be developed, and a concurrent function or procedure call can be made from within a guarded block. This alternative though requires developing a separate package, and if the system being modeled requires many little functions or procedures, the readability of the program quickly becomes cumbersome. The second alternative is to use a process statement where signal assignments are processed sequentially. However, the sensitivity list of a process statement does not constitute a guard expression, and a null transaction cannot be assigned from within a process directly. It is possible though to nest a process statement within a guarded block statement (Arm89). The sensitivity list of the nested process statement contains the simple name GUARD and control of the signal drivers within the process are as shown in the following example:

```
A: block(guard expression)
  begin
    process(GUARD)
    begin
      if GUARD then
        X <= '1';
      else
        X <= null;
      end if;
    end process;
  end block A;
```

In this example, block A has some Boolean-valued guard expression that when True, enables the nested process statement. Within the process statement, the implicit signal GUARD is tested, and if true, the signal X is assigned the value '1'. If GUARD is not true, the signal X will receive null disconnecting that driver from the bus resolution function defined for X.

The above method of nesting a process statement within a block statement is used extensively throughout the FPASP VHDL simulator to control the signal drivers of registers.

5.5.1 Two Phase Clock Model: The sequencing of all internal FPASP events are controlled by an external clock. The external clock controls when registers place their contents on the busses and when the registers load new contents from the busses, when the microsequencer places a new ROM microword in the pipeline and when the pipeline outputs its contents onto the control signal paths, and when the processing components accept data from the busses for computations. The FPASP uses the MS flip-flop exclusively for its registers, and this type of flip-flop requires a two phase, non-overlapping clock for proper operation. The FPASP is designed to operate with a clock frequency of 25MHz which equates to a period of 40ns.

VHDL facilitates modeling this type of a clock, and the process shown below illustrates how the two cycle external clock for the VHDL simulator was modeled.

```
CLOCK: process
begin
  PRECHARGE <= '1' after 2ns,
               '0' after 10ns;

  PHI1      <= '1' after 2ns,
               '0' after 10ns;

  PHI2      <= '1' after 12ns,
               '0' after 38ns;

  wait for 40ns;
end process CLOCK;
```

The three signals, PHI1, PHI2, and PRECHARGE, are signals of type Bit. The values these signals can assume are either '0' or '1'. At the beginning of the simulation cycle, the signals

PHI1 and PRECHARGE are driven with the value '1'. After a 10ns delay, the signals PHI1 and PRECHARGE are driven with the value '0'. After another 2ns delay, the signal PHI2 is driven with the value '1' which is held for a period of 26ns after which it is driven with the value '0'. The process waits for the full 40ns clock period after which the process repeats. The process will repeat continuously until an explicit request to terminate is received. The three clock signals are local to the architecture containing the process CLOCK, however, these signals can be listed in the port map of an instantiated component making them visible to a component for timing control.

The process CLOCK generates the simulated clock signals mentioned above which are used throughout the FPASP model to control the sequencing of the VHDL simulator's design entities.

5.5.2 Upper Data Path Registers Model: Two design entities were developed to model the FPASP registers. One design entity, ASP_UPPER_REGS, models the registers on the upper data path and the other, ASP_LOWER_REGS, models the registers on the lower data path. As stated previously, the registers modeled in each of these design entities consists of the 25 general purpose registers, the MBR, the MAR, the memory address pointers and their increment registers, and the incrementable registers. Both of the design entities are identical to one another with the exception of their port lists in the entity declarations.

The port list of ASP_UPPER_REGS identifies the upper **A** bus, the upper **B** bus, the upper **C** bus, the upper **D** bus, the upper **E** bus, the upper address bus, Φ_1 and Φ_2 from the external clock, and the upper ROM control word as shown below:

```
entity AspUpperRegs is
port (UA_BUS   : out    BUS_TYPE bus;
      UB_BUS   : out    BUS_TYPE bus;
      UC_BUS   : in     BUS_TYPE bus;
      UD_BUS   : inout  BUS_TYPE bus;
      UE_BUS   : inout  BUS_TYPE bus;
      UP_ADD   : out    BUS_TYPE bus;
      PHI1     : in     BIT;
      PHI2     : in     BIT;
      UCW      : in     UPPER_ROM_WORD);
end AspUpperRegs;
```

All of the busses are of the type BUS_TYPE which is a 32 bit wide resolved signal. The two clock inputs, PHI1 and PHI2, are of the type BIT which can assume the values of either '0' or '1'. The signal UCW is the upper ROM control word of type UPPER_ROM_WORD which is a BIT vector 64 bits wide. The signals identified in the port list are the only signals through which the entity ASP_UPPER_REGS communicates to the outside.

An Entity Structure Diagram of the ASP_UPPER_REGS design entity is shown in Figure 5.2. The Entity Structure Diagram syntax is given in Appendix B. The registers that drive their outputs onto the upper **A** bus are the 25 general purpose registers, the incrementable registers, the memory address pointers, and the MBR. The Upper A Bus Select micro–field of the upper ROM control word determines which register is driving its contents onto the upper **A** bus at any one time. A single block statement with a nested process statement was developed to emulate this activity as shown below.

```
UREG_ABUS: block(PHI1 = '0')
  begin
    process(GUARD)
    begin
      if GUARD then
        case Upper_A_Bus_Select is
          when "00001" => UA_BUS <= UREG_1;
          when "00010" => UA_BUS <= UREG_2;
                        :
          when "11001" => UA_BUS <= UREG_25;
          when "11010" => UA_BUS <= UINC1;
          when "11011" => UA_BUS <= UINC2;
          when "11100" => UA_BUS <= UINC3;
          when "11101" => UA_BUS <= UAPT;
          when "11110" => UA_BUS <= UBPT;
          when "11111" => UA_BUS <= UMBR;
        end case;
      else
        UA_BUS <= null;
      end if;
    end process;
  end block UREG_ABUS;
```

The block is guarded with the single Boolean-valued expression of (PHI1 = '0') which enables

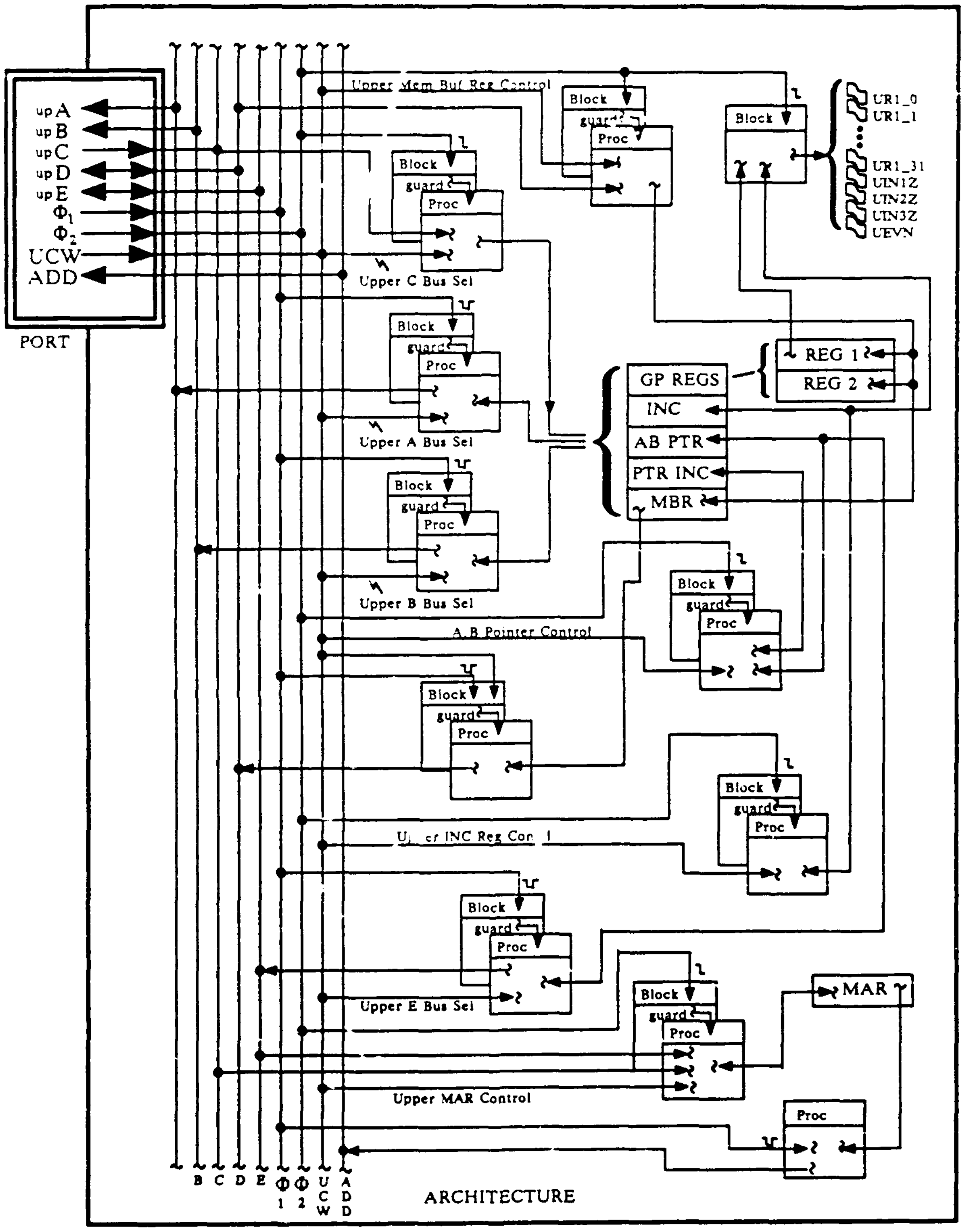


Figure 5.2. ASP_UPPER_REGS Entity Structure Diagram

the block when ever PHI1 equals zero thereby allowing the nested process to decode the Upper A Bus Select micro–field, and drive the contents of the appropriate register onto the **A** bus. Whenever PHI1 equals one, the block is disabled and the value 'null' is driven onto the bus turning the drivers off.

A similar block statement with a nested process statement was developed to support driving the contents of the registers onto the **B** bus. The Upper B Bus Select micro–field is decoded to determine which register is selected to drive the bus. The other exception is rather than allowing the contents of the two memory address pointers to be driven onto the **B** bus, the upper outputs of the floating point multiplier and adder are selected in their place.

A single process statement is provided that places the contents of the MAR on the Upper Address Bus at the beginning of each simulation cycle. The process consists of a wait statement that waits for the falling edge of PHI1. With the falling edge of PHI1, a signal assignment statement drives the contents of the MAR onto the address bus.

A block statement with a nested process is used to decode the Upper E Bus Select micro–field and drive either the contents of A memory address pointer or the B memory address pointer onto the **E** bus. The block statement guard expression is True when PHI1 is zero and a case statement is used to decode the micro–field.

The only other block statement that is enabled during the period that PHI1 equals zero is the block that drives the contents of the MBR onto the **D** bus. Here too, a process statement is nested inside the block statement. There is no explicit micro–field which controls the whether the MBR drives it contents onto the **D** bus or not. Rather, this condition is implied by the Upper Memory Write Enable Bar micro–field. If the condition is set for the FPASP to write to external memory, the MBR must place its contents onto the **D** bus. Likewise, if the condition is set for the FPASP to read from external memory, the value on the **D** bus must be loaded into the MBR. An alternative method for decoding the single bit Upper Memory Write Enable Bar micro–field is to test the bit in the guard expression of the block statement as shown:

```
DRIVE_UDBUS: block(PHI1 = '0' and UCW(43) = '0')
  begin
    process(GUARD)
    begin
      if GUARD then
        UD_BUS <= UMBR;
      else
        UD_BUS <= null;
      end if;
    end process;
  end block DRIVE_UDBUS;
```

All block and process statements so far have functioned to drive the value contents of registers onto the busses. The next block statement loads the values on the busses into the registers, and is implemented in a slightly different manner.

A single block statement is used to control the GUARD signal of several nested process statements that load the values on the busses into the registers. The guard condition for this block is True when PHI2 equals zero and the signal is not stable indicating this is the falling edge. This approach is justified when viewing Figure 3.2.

The first nested process statement loads the value on the **C** bus into a register selected by decoding the Upper C Bus Select micro–field. Each register in the FPASP hardware simulator is a signal of the kind **register** which means the signal retains its value when all of its drivers are turned off. This is necessary in cases where it is possible to load a register in more than one way. Registers identified as being driven by more than one source are the general purpose registers 1 and 2, the incrementable registers, and the memory address pointers. These registers have their drivers turned off with a null transaction if they are not selected. Decoding of the Upper C Bus Select micro–field is performed using a case statement as before.

The second nested process statement loads the value on the **D** bus into either general purpose registers 1 or 2, or the MBR. The determination is made by decoding the Upper Memory Buffer Registers Control micro–field. If the guard condition is False, the drivers to these registers are turned off using null transactions.

The third nested process statement performs the function of incrementing the memory address pointer registers, loading the memory address pointer increment registers, or both. The A,B Pointer Controls micro–field is decoded using a case statement to determine which function the process statement needs to perform. Either the A memory address pointer increment register, B memory address pointer increment register, or both can be loaded from the **C** bus. Incrementing either the A memory address pointer or the B memory address pointer is performed by adding the value contained in each increment register to its respective pointer register. If a function is not selected, the drivers for each of the registers is turned off using a null transaction.

The fourth nested process statement performs the functions of incrementing the incrementable registers. The Upper Incrementable Registers Control micro–field is decoded using a case statement. The appropriate incrementable register has the value of one added to its current value if selected, otherwise the drivers are turned off.

The fifth and last nested process statement loads the values on the **C** bus or the **E** bus into the MAR, or it increments the MAR by one. The Upper Memory Address Register Control micro–field is decoded using a case statement to select the appropriate function. If no function is selected, the drivers to the MAR are turned off.

In addition to driving register contents onto the busses and loading bus values into the registers, this architecture must also monitor the conditions of several registers so that appropriate flags may be set. The bit settings of the four least significant bits and the four most significant bits of general purpose register 1 must be monitored, the incrementable registers must be tested for zero, and the value on the **C** bus be tested to determine whether it is odd or even.

The simple process statement shown below monitors the bit settings of general purpose register 1.

```
MONITOR_UREG1: process(UREG_1)
  begin
    UR1_0_IN    <= UREG_1(0);
    UR1_1_IN    <= UREG_1(1);
    UR1_2_IN    <= UREG_1(2);
    UR1_3_IN    <= UREG_1(3);
    UR1_28_IN   <= UREG_1(28);
    UR1_29_IN   <= UREG_1(29);
    UR1_30_IN   <= UREG_1(30);
    UR1_31_IN   <= UREG_1(31);
  end process MONITOR_UREG1;
```

The sensitivity list of the process contains the simple name of the register, UREG_1, which enables the process whenever a new value is loaded. The process then slices out the appropriate bits from the register and assigns the bit values in the input side of their respective flags. Three similar process statements monitor the incrementable registers. The sensitivity lists of the process statements each contain one of the names of the three incrementable registers in the upper data path. Any change to the value contained in the incrementable registers enables its respective monitor process which calls a function passing the value of the register in the argument. The result of the function call is loaded into the input side of the respective flags. A final process statement similarly monitors the value on the C bus. The sensitivity list of this process statement contains the name of the C bus, and any change to its value enables the process where the least significant bit of the bus is inverted and loaded into its flag. A block statement is provided in the architecture which drives the values of from the input side of the flags to the output side. The guard expression for this block statement is True when PHI2 equals zero and the signal is not stable. When the guard expression is True, concurrent signal assignments transfer the data to the output side of the flags making their values available for the next clock cycle.

5.5.3 Lower Data Path Registers Model: The registers of the lower data path are mirror images of the registers on the upper data path. Consequently, the architecture developed to emulate the behavior of these registers is identical to that given above for the upper data path with two exceptions. The first exception is that the signals declared in the entity declaration port list are those particular to the lower data path. The second exception is that the bit settings of lower general purpose register one are not flagged in the lower data path.

5.5.4 Control Section Model: The components that comprise the FPASP control section are the CAR, MUX, condition MUX, stack, branch logic, and XROM/LPROM. The FPASP control section is called the Microsequencer. The CAR is a 10 bit wide MS flip-flop register that holds the addresses used to index into the XROM/LPROM microcode store. The MUX selects one of four sources that are used to update the address held in the CAR. Two of these sources take the next address directly from the microcode word. One source comes from the stack when an address is popped to return from a subroutine call. The last source is from the Mapping ROM which is not supported in this simulator at this time. The Branch Logic and the Condition MUX work together to test the various system flags used for conditional branches and subroutine calls. The Stack is used to hold addresses used for returns from subroutines.

The architecture of the design entity developed to model the Microsequencer behavior is shown in Figure 5.3. The architecture has three process statements that emulate its functions.

The first process models the functions of the CAR, the MUX, the Stack, the Branch Logic, and the XROM/LPROM. The second process statement models the Microsequencers pipeline, and splits the Pipeline output into the Upper and Lower ROM control words that are routed out to the rest of the simulator. The third process statement models the Condition Mux.

The port list of the entity declaration lists both clock signals, PHI1 and PHI2, and the Upper and Lower Rom Control words as shown below:

```
entity MICRO_SEQ is
port (PHI1                         : in    BIT;
      PHI2                         : in    BIT;
      UPPER_CONTROL_WORD           : inout UPPER_ROM_WORD;
      LOWER_CONTROL_WORD           : inout LOWER_ROM_WORD);
end MICRO_SEQ;
```

The first process statement, at the beginning of the simulation, initializes the contents of the XROM/LPROM from the external *.trans* disk file and sets the contents of the CAR to zero. After initializing the XROM/LPROM, the process waits for the rising edge of the signal PHI1. With the rising edge of PHI1, the XROM/LPROM is indexed using the CAR and the new microcode word is driven into the input of the Pipeline register. Next, the Branch Control micro–field of the microcode word on the output side of the Pipeline register is examined. If a conditional branch is requested (the default), the output of the Condition Mux process is tested, and if True, the literal address field of the microcode word is used for the next address to be loaded into the CAR. Otherwise, the CAR is incremented by one. If a conditional call to a subroutine is requested, the output of the Conditional Mux process is tested, and if True, the current value in the CAR is incremented by one and pushed onto the Stack, the Stack Pointer incremented, and the literal address field of the microcode word is used for the next address. Otherwise, the CAR is incremented by one. If a conditional return from a subroutine is requested, the output of the Conditional Mux

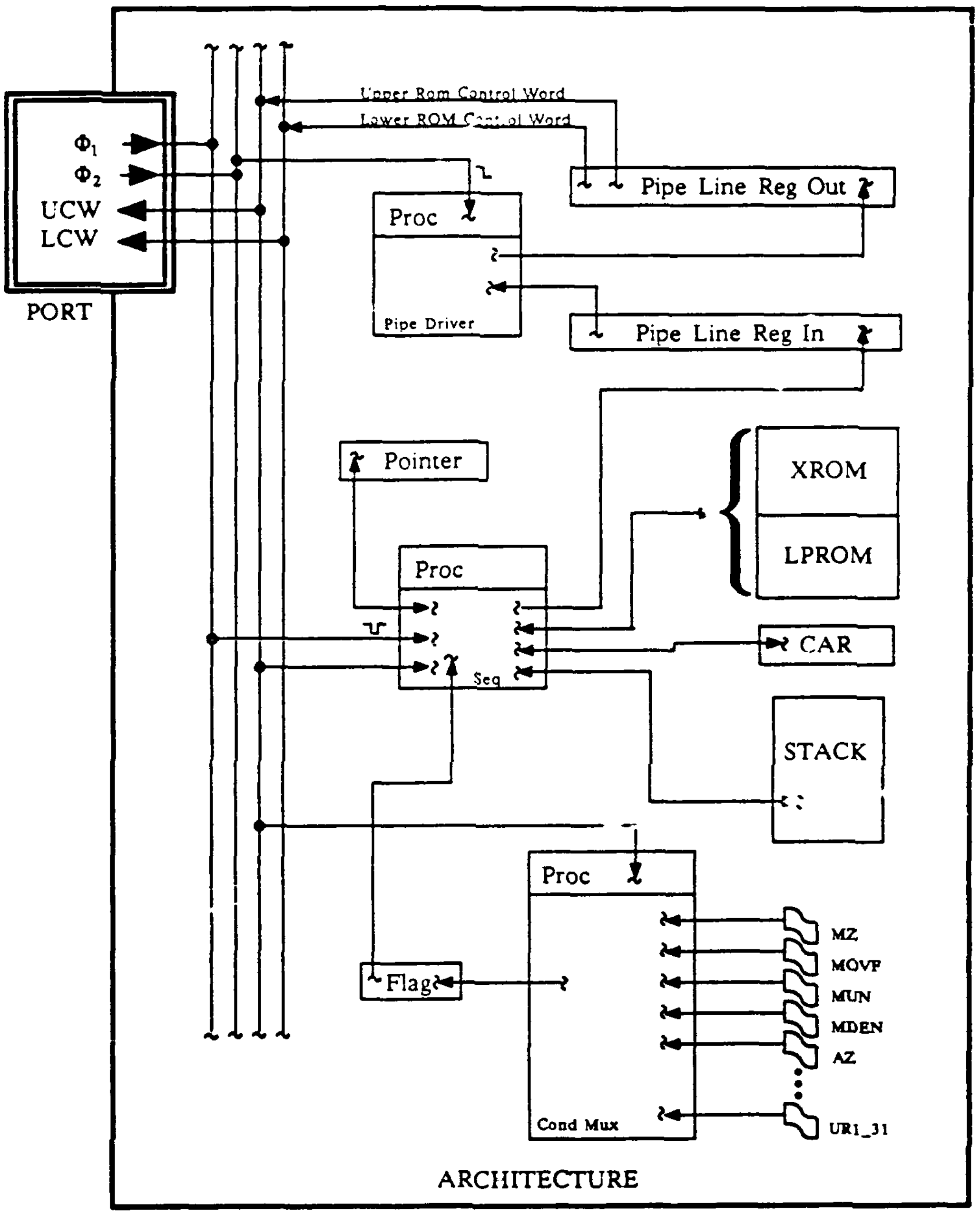


Figure 5.3. Microsequencer Entity Structure Diagram

process is again tested, and if True, the Stack Pointer is decremented and a return address is popped from the Stack and loaded into the CAR.

The second process statement waits for the falling edge of the signal PHI2 at which time the microcode word at the input of the Pipeline is assigned to the Pipeline output.

The third process statement models the Condition Mux as shown in the abbreviated example below.

```
MUX_CONDITION: process(UPPER_CONTROL_WORD)
variable FLAG: ALU_BUS_BIT;

  begin
  case MUX_SLICE is
    when "000000" => FLAG := '0';
    when "000001" => FLAG := '0';
    when "000010" => FLAG := '0';
    when "000011" => FLAG := MZ_OUT;
    when "000100" => FLAG := MOVF_OUT;
                     :
    when "011100" => FLAG := UR1_0_OUT;
                     :
    when "111110" => FLAG := UR1_30_OUT;
    when "111111" => FLAG := UR1_31_OUT;
  end case;
  MUX_FLAG <= FLAG;
  end process MUX_CONDITION;
```

The process statement is sensitive to changes in the UPPER_CONTROL_WORD. MUX_SLICE is the Conditional Mux Select micro–field of the microcode word. Every simulation cycle, the output of the Pipeline register is driven into the signal UPPER_CONTROL_Word which enables the process statement. Once the process statement is enabled, the local variable FLAG is either set to the value of an internal FPASP flag, or to a fixed value as indicated by the Conditional Mux Select micro–field. The value assigned to the variable FLAG is then used to drive the signal MUX_FLAG which is visible to the first process statement which tests this signal for conditional branches and subroutine calls.

5.5.5 Upper ALU/Linear Shifter Combination Model: The functions of the components, ALU and Linear Shifter, were combined into one design entity. The ALUs perform simple arithmetic or logic functions on each of the 32 bit integer data paths. Incorporated into each ALU cell is a linear shifter which can perform a left, right, or no shift on the data outputed from the ALU with a variety of sources for the bit to be shifted in. The ALU and linear shifter produce five flags for conditional branching. These flags are held in MS flip-flops so they can be used until reset.

The port list of the entity declaration for the design entity modeling the upper ALU/Liner Shifter lists the upper **A** bus, the upper **B** bus, the upper **C** bus, PHI1 and PHI2, and the Upper Control Word as input and output signals as shown below.

```
entity UPPER_ALU is
port (UA_BUS  : in     BUS_TYPE bus;
      UB_BUS  : in     BUS_TYPE bus;
      UC_BUS  : inout  BUS_TYPE bus;
      PHI1    : in     BIT;
      PHI2    : in     BIT;
      UCW     : in     UPPER_ROM_WORD);
end UPPER_ALU;
```

The architecture of the design entity UPPER_ALU is depicted in Figure 5.4. This architecture contains a single block statement with a nested process statement that emulates the functions of both the ALU and the Linear Shifter.

The guard condition of the block becomes True when PHI1 is zero and stable which enables the nested process statement. The sensitivity list of the process statement in addition to the implicit signal GUARD also contains the name of the upper **A** bus and the upper **B** bus. The functionality of the ALU and the Linear Shifter are not sequenced by any clock action. Both components are composed of combinational logic circuits that are tied to both the input busses, and any values placed on the busses are processed even if the results are ignored. Placing the names of both the upper input busses in the sensitivity list insures the architecture models this activity. It would seem more appropriate to place the names of the input busses using appropriate Boolean-valued signal attributes in the guard expression of the block thereby enabling the block when ever changes occur on the busses. This would work except for the fact that the driver for the **C** bus would be

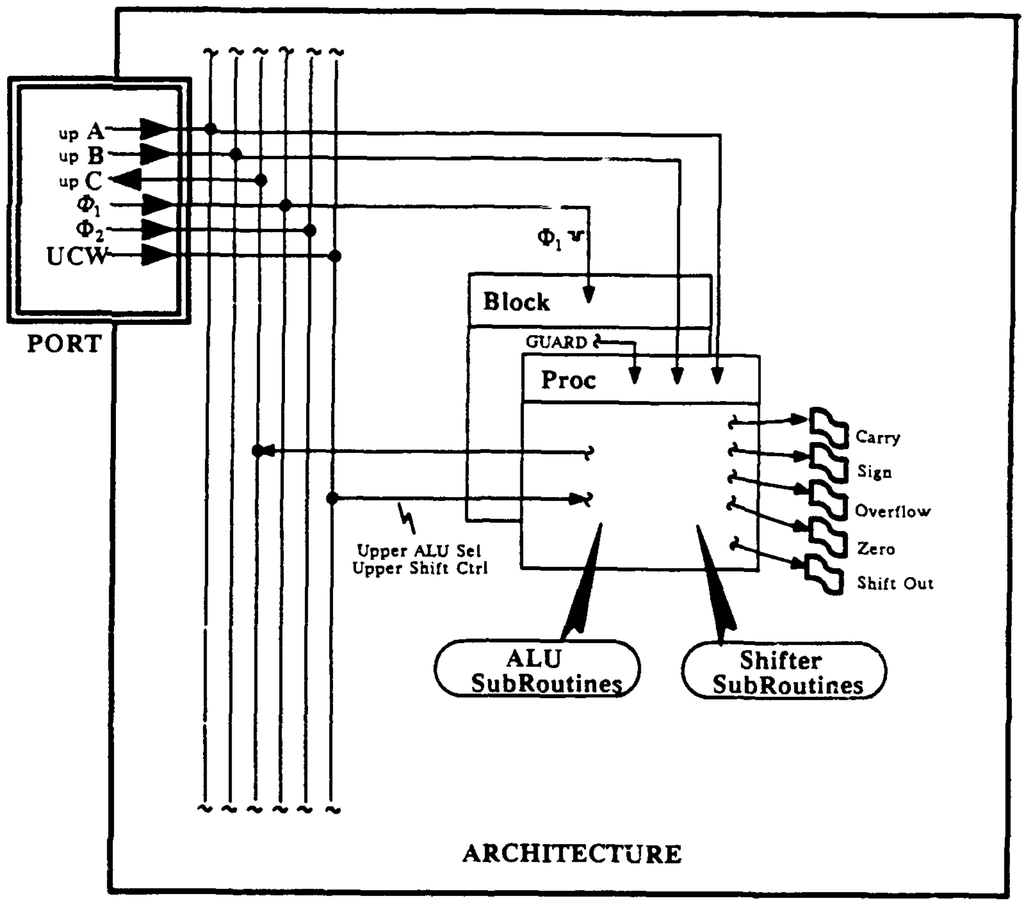


Figure 5.4. ALU/Shifter Entity Structure Diagram

turned off before the data output could be loaded into a receiving register. Hence, the need to ensure the block is enabled for the entire time that PHI1 is zero which includes the time at which the falling edge of PHI2 occurs.

The ALU is capable of performing 16 different functions on 32 bit integers which can be reviewed in Appendix A. Selection of any one of the functions is done by decoding the Upper ALU Select micro-field of the upper ROM control word. To model the 16 functions of the ALU, 16 individual subroutines were written and placed in a separate package. The majority of the subroutines perform simple functions like ORing or ANDing two inputs together and checking the result for zero. The arithmetic functions operate at the bit level of the input integers and set the appropriate Sign, Carry, Zero, and Overflow flags during or after computations. The decoding of the Upper Alu Select micro-field is performed using a case statement. The outputs of the ALU subroutines are placed in a local variable that is used as the input to the Linear Shifter.

The Linear Shifter is capable of performing 15 different functions on 32 bit integers which can also be reviewed in Appendix A. Selection of one of the Linear Shifters functions is done by decoding the Upper Shift Control micro-field in the upper Rom control word. As was done for the ALU, the 15 functions of the Linear Shifter were modeled using subroutines which were placed in a separate package. The decoding of the Upper Shifter Control micro-field is performed using a case statement. The outputs of the Shifter subroutines are used to drive the **C** bus directly.

The use of subroutines to model the functions of both the ALU and the Linear Shifter is consistent with developing a model of a digital system at the register level. It was brought out earlier in this thesis that the circuit level design of the FPASP is still ongoing and therefore subject to change. Once the circuit level designs are finalized, they can be modeled with VHDL design entities and instantiated into the upper ALU/Linear Shifter architecture.

An additional block statement is used in architecture of the upper ALU/Linear Shifter to drive the values of the flags from the input side of the flags where they were placed by the various routines, to the output side where they can be used for conditional branching in the next clock cycle. The guard condition of the block enables the block with the falling edge of the clock signal PHI2 where guarded concurrent statements assign the values to the flag outputs.

5.5.6 Lower ALU/Linear Shifter Combination Model: The ALU/Linear Shifter combination on the lower data path is a mirror copy of the ALU/Linear Shifter combination on the upper data path. The port list for the lower combination is identical to that of the upper combination with the exception that signals referenced are particular to the lower data path. In a similar fashion, the architecture of the lower ALU/Linear Shifter architecture uses subroutines to model the behavior of the ALU and the Linear Shifter. These subroutines are placed in a separate package.

5.5.7 Literal Inserter Model: The Literal Inserter takes its input from the least significant 16 bits (Literal Field) of the lower ROM word and places those bits on either the upper or lower **A** bus in one of 12 different ways. The different ways that the Literal Field of the lower ROM word can be placed on the **A** busses can be reviewed in Appendix A. Decoding the Literal Inserter Control micro-field of the upper ROM word determines which way the Literal Field is placed on the busses.

The port list of the design entity that models the Literal Inserter identifies the upper and lower **A** busses, both PHI1 and PHI2 clock signals, and both the upper and lower ROM controls words as shown below.

```
entity LIT_INSERT is
port (UA_BUS   : inout   BUS_TYPE bus;
      LA_BUS   : inout   BUS_TYPE bus;
      PHI1     : in      BIT;
      PHI2     : in      BIT;
      UCW      : in      UPPER_ROM_WORD;
      LCW      : in      LOWER_ROM_WORD);
end LIT_INSERT;
```

The architecture modeling the behavior of the Literal Inserter uses a process statement nested inside a block statement as shown in Figure 5.5.

The guard condition of the block statement is True when the bit pattern of the Literal Inserter Control micro-field is all zeros, and when PHI1 is zero and Stable. The sensitivity list of the nested process statement lists the implicit signal GUARD as the only signal it is sensitive to. Decoding of the Literal Inserter Control micro-field is performed using a case statement. The actual behavior of

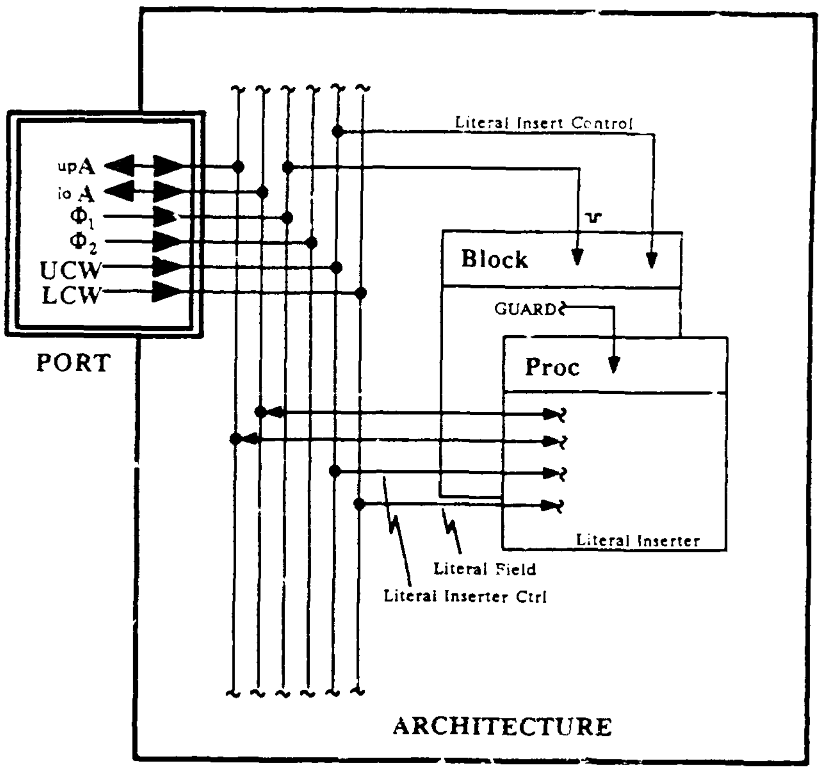


Figure 5.5. Literal Inserter Entity Structure Diagram

the Literal Inserter is modeled using sequential assignment statements within the process statement as shown below.

```
INSERT: block((INS_CONT = "0000") and (PHI1 = '0'))
begin
  process(GUARD)
  begin
  variable TEMP_BUS01 : BUS_TYPE;
  variable TEMP_BUS02 : BUS_TYPE;
  begin
    if GUARD then
      case INS_CONTROL is
        when "0100" =>
                    TEMP_BUS01(31 downto 0):= HALF_ZERO;
                    for I in 48 to 63 loop
                      if LCW(I) = '1' then
                        TEMP_BUS01(63-I):= '1';
                      else
                        TEMP_BUS01:= '0';
                      end if;
                    end loop;
                    TEMP_BUS02:= TEMP_BUS01 and LA_BUS;
                    LA_BUS <= TEMP_BUS02;
        when "0101" =>
                    ⋮
      end case;
    else
      UA_BUS <= null;
      LA_BUS <= null.
    end if:
  end process;
end block INSERT:
```

The width of the Literal Field is only 16 bits and can be placed in either the upper half or the lower half of the bus. In the example above. the upper half of the variable TEMP_BUS01 is zeroed, and the Literal Field of the Lower Control Word is read in one bit at a time into the lower half of the temporary variable. The temporary variable is then ANDed with whatever is on the LA_BUS, and the results are driven back onto the bus. All the variations of inserting the Literal Field onto the **A** busses are modeled in a similar fashion.

5.5.8 Bus Ties Model: The Bus Ties provide an easy method of electrically connecting selected busses together to facilitate transferring data between the upper and lower data paths. This means that data is free to travel in both directions across the switch. The busses that are provided with bus ties are the upper and lower **A** busses, upper and lower **B** busses, upper and lower **C** busses, and the upper and lower **E** busses.

The design entity developed to model the behavior of the Bus Ties has the port list shown below.

```
entity BUS_TIES is
port (UA_BUS   : inout   BUS_TYPE bus;
      LA_BUS   : inout   BUS_TYPE bus;
      UB_BUS   : inout   BUS_TYPE bus;
      LB_BUS   : inout   BUS_TYPE bus;
      UC_BUS   : inout   BUS_TYPE bus;
      LC_BUS   : inout   BUS_TYPE bus;
      UE_BUS   : inout   BUS_TYPE bus;
      LE_BUS   : inout   BUS_TYPE bus;
      PHI1     : in      BIT;
      PHI2     : in      BIT;
      UCW      : in      UPPER_ROM_WORD;
      LCW      : in      LOWER_ROM_WORD);
end BUS_TIES;
```

The architecture developed for the design entity modeling the Bus Ties has four block statements each with a nested process statement as shown in Figure 5.6.

Modeling the switch action for the **A**, **B**, and **E** busses is straight forward because all of these busses are precharged busses. The **C** bus is not precharged and modeling the behavior of its switch was handled in a different manner.

There are four independent micro–fields controlling the actions of the four bus ties. The guard condition of the block statement for the A Bus Tie is True when PHI1 is zero and when A Bus Tie micro–field bit of the lower ROM word is set to '1'. The sensitivity list of the nested process contains the name of the implicit signal GUARD and the upper and lower **A** busses. This insures that any changes on either of the **A** busses is reflected in their values. Inside the process statement, the values on both the **A** busses are ANDed together, and the result is driven onto both

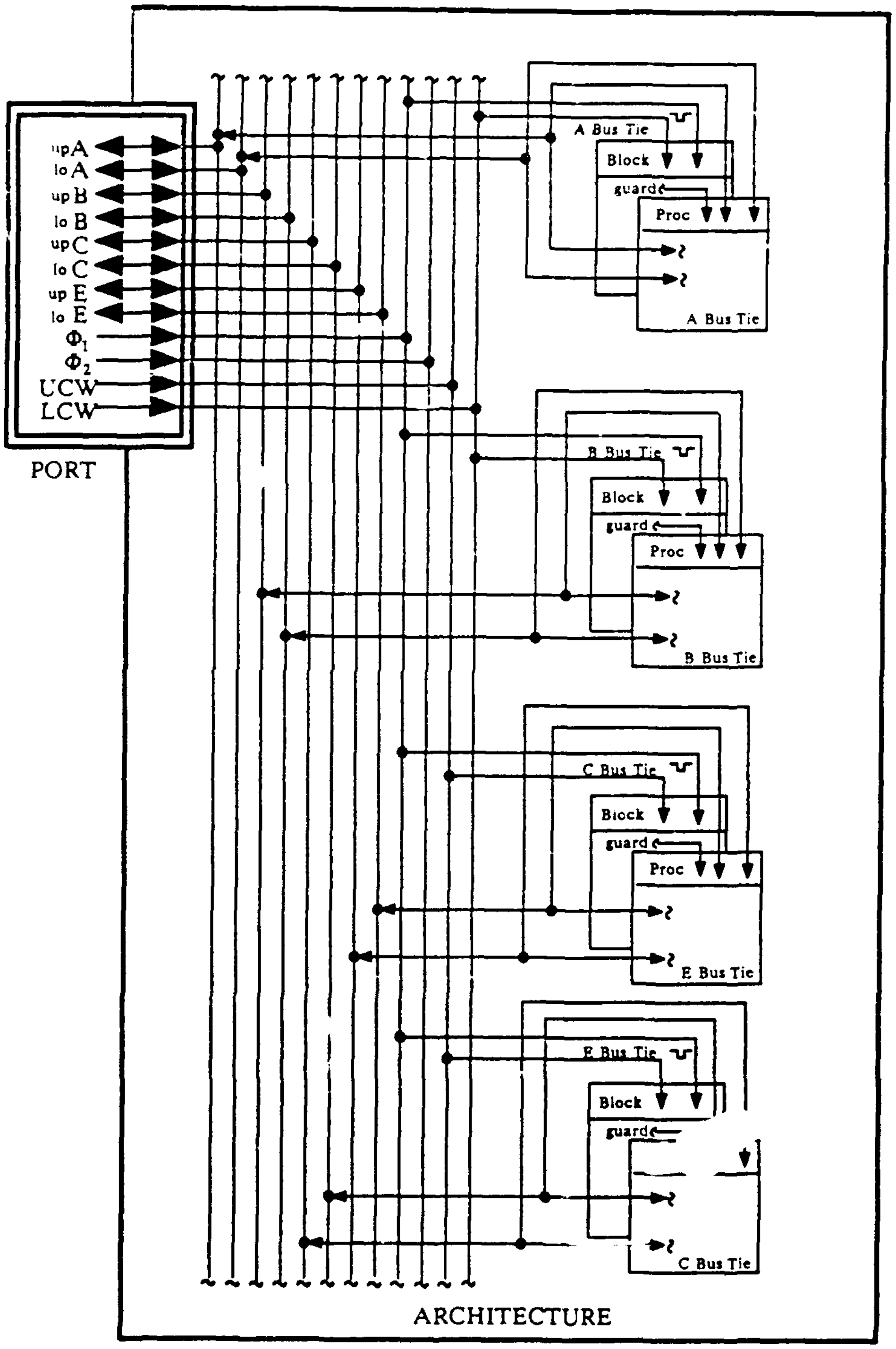


Figure 5.6. Bus Ties Entity Structure Diagram

the busses. The block/process combinations that model the bus tie action for the **B** and **E** busses are similar with the exception that the appropriate micro–fields are tested in the guard expression of their respective blocks.

Modeling the C Bus Tie required a different approach than was used with the precharged busses. The architecture developed to model the action of the C Bus Tie models the behavior and not necessarily how it is implemented in the FPASP. As before, the guard condition of the block statement is True when PHI1 is zero and the C Bus Tie micro–field bit of the upper ROM control word is '1'. The sensitivity list of the nested process statement lists the implicit signal GUARD as well as the upper and lower **C** busses. However, within the process statement, the values on the **C** busses can not be ANDed together if the C Bus Tie is selected. Judicious care was taken in developing the various design entities that model the components of the FPASP to insure that if the **C** busses are not driven with a value, that they are assigned a null transaction to turn the drivers off. This action assigns the default value of all Zs, or high impedance, to each driver that is turned off. If a C Bus Tie is requested with only one bus active with a valid value, the other bus will be in the high impedance state. This condition is tested for in the process statement, and the bus in the high impedance state is assigned the value of the active bus. If both busses are active, both busses not in the high impedance state, the values on the busses are ANDed together and the result driven out onto each of the busses.

5.5.9 Function ROM Model: The Function ROM is a repository for seed values used in recursive microcode routines and programs. The Function ROM consists of seven pages of 32 x 5 bit words. Indexing into the Function ROM requires the combination of data values on the upper **B** bus and the Function ROM Select micro–field of the upper ROM word. Data read from the Function ROM is placed on the upper **C** bus.

The port list for the entity declaration of the Function ROM model is shown below, and consists of the upper **B** bus, the upper **C** bus, PHI1 and PHI2, and the upper ROM Control Word.

```
entity FUNCTION_ROM is
port (UB_BUS  : in    BUS_TYPE bus;
      UC_BUS  : out   BUS_TYPE bus;
      PHI1    : in    BIT;
```

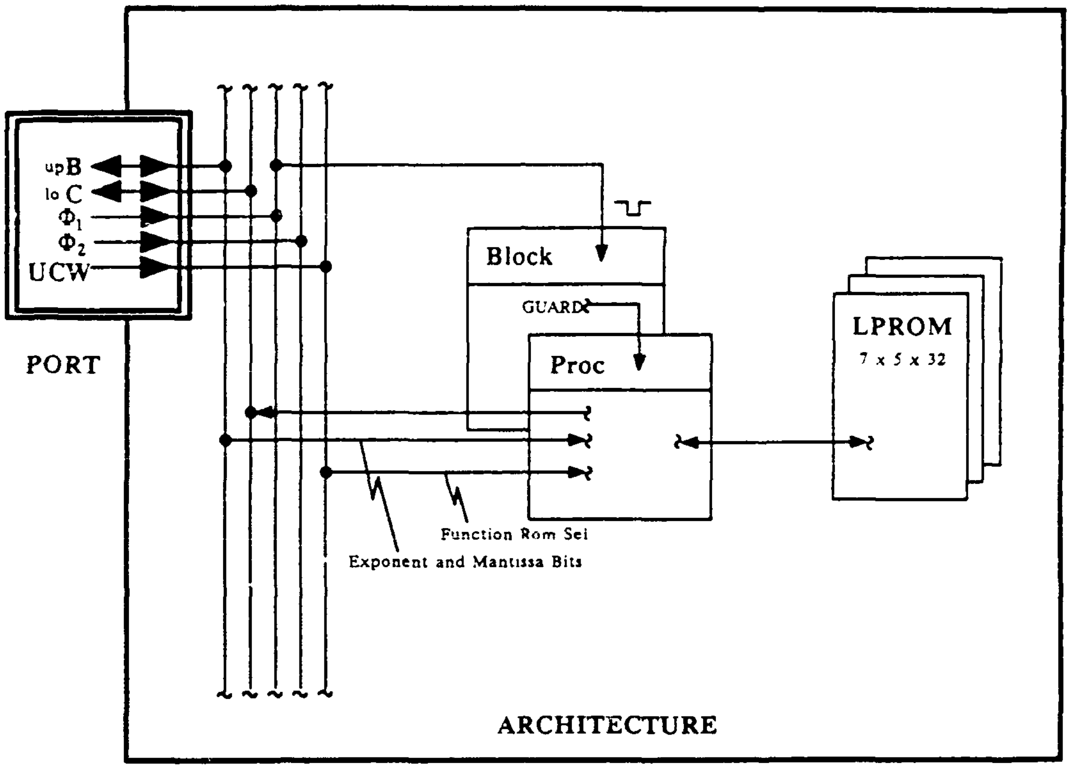


Figure 5.7. Function ROM Entity Structure Diagram

```
        PHI2      : in      BIT;
        UCW       : in      UPPER_ROM_WORD);
    end FUNCTION_ROM;
```

The architecture developed to model the behavior of the Function ROM has a single block statement with a nested process statement which is shown below in Figure 5.7.

The guard expression of the block statement is True when PHI1 is zero thereby enabling the nested process statement. The sensitivity list of the process statement contains the name of the implicit signal GUARD. The decoding of the Function ROM Select micro-field of the upper ROM Control Word is performed using a case statement. The Function ROM Select micro-field designates which of the seven Function ROM pages is to be read from. The 20^{th} down to the 16^{th} bit positions of the value on the upper B bus are used to determine which 5 bit word is read from a selected page. The 20^{th} bit position is the least significant bit position of the exponent of a double precision IEEE floating point number. Bit positions 19 down to 16 are the 5 most significant bits of the mantissa.

The process statement slices the 20th down to 16th bits from the value on the upper **B** bus and converts the bit string into an integer. The case statement then tests each of the possible Function ROM Select micro-field combinations. Each of the seven Function ROM pages is associated with one of the conditional tests of the case statement. If the proper page is found, the integer converted bit-slice from the upper **B** bus is used to index into the array and the indexed value is driven onto the upper **C** bus. If the Function ROM was not selected, the **C** bus signal associated with this architecture is driven with a null value to turn the driver off.

5.5.10 Barrel Shifter Model: The Barrel Shifter is implemented in the FPASP to provide the capability to perform circular left integer shifts that are greater than one. The Barrel Shifter can be programmed to shift from either directly from the microcode, or from the Barrel Shifter Set-up register. The Barrel Shifter Set-up register can be loaded from the five LSBs of the **C** bus. This option provides the capability to set the shift amount from a previous calculation.

The port list for the entity declaration of the Barrel Shifter is shown below, and consists of the lower **A** and **C** busses, PHI1 and PHI2, and the lower ROM Control Word.

```
entity BARREL_SHIFTER is
port (LA_BUS   : in    BUS_TYPE bus;
      LC_BUS   : out   BUS_TYPE bus;
      PHI1     : in    BIT;
      PHI2     : in    BIT;
      LCW      : in    LOWER_ROM_WORD);
end BARREL_SHIFTER;
```

The architecture developed to model the behavior of the Barrel Shifter is shown in Figure 5.8. The architecture has a single block statement that controls the loading of the Barrel Shifter Set-up register, and a nested process statement to model the behavior of the shifter.

The guard condition of the single block statement is True on the falling edge of PHI2, and when MSB of the Barrel Shifter Set- up micro-field is '1'. When the guard expression is True, the five LSBs from the **C** bus are loaded into the Barrel Register. The guard expression of the block

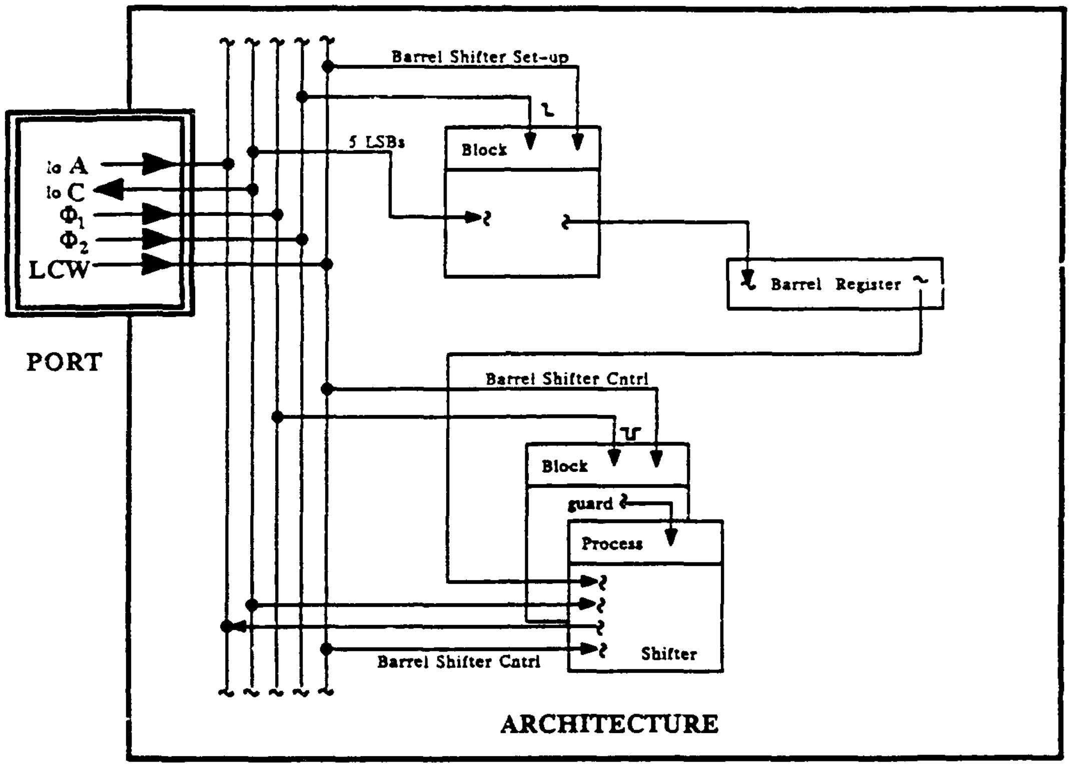


Figure 5.8. Barrel Shifter Entity Structure Diagram

with the nested process is True whenever *PHI1* is '0', and whenever the Barrel Shifter micro-field is not "00000". The sensitivity list of the process statement contains the name of the implicit signal GUARD. The body of the process statement further decodes the Barrel Shifter Control micro-field, and performs the required number of left circular shifts from the appropriate control input. The final output is placed back on the lower **C** bus.

5.5.11 Floating Point Adder/Subtractor Model: The Floating Point Adder/Subtractor is a single macro-cell developed with combinational logic to perform addition and subtraction with double precision IEEE floating point numbers with a latency of two clock cycles. For single floating point additions or subtractions, an extra clock cycle is required to load the input registers to the adder. The latency of two clock cycles is obtained with repeated floating point operations in tight loops where the adder can both read in new inputs and drive out previous results in the same clock cycle.

The port list for the entity declaration of the Floating Point Adder/Subtractor is shown below and declares the upper and lower **A** busses, the upper and lower **B** busses, the upper and lower **C** busses, clock signals *PHI1* and *PHI2*, and the upper ROM Control Word.

```
entity FP_ADDER is
port (UA_BUS  : inout  BUS_TYPE bus;
      LA_BUS  : inout  BUS_TYPE bus;
      UB_BUS  : inout  BUS_TYPE bus;
      LB_BUS  : inout  BUS_TYPE bus;
      UC_BUS  : inout  BUS_TYPE bus;
      LC_BUS  : inout  BUS_TYPE bus;
      PHI1    : in     BIT;
      PHI2    : in     BIT;
      UCW     : in     UPPER_ROM_WORD);
end FP_ADDER;
```

The architecture developed to model the behavior of the Floating Point Adder/Subtractor has several block/process statement combinations, process statements, and block statements as shown below in Figure 5.9.

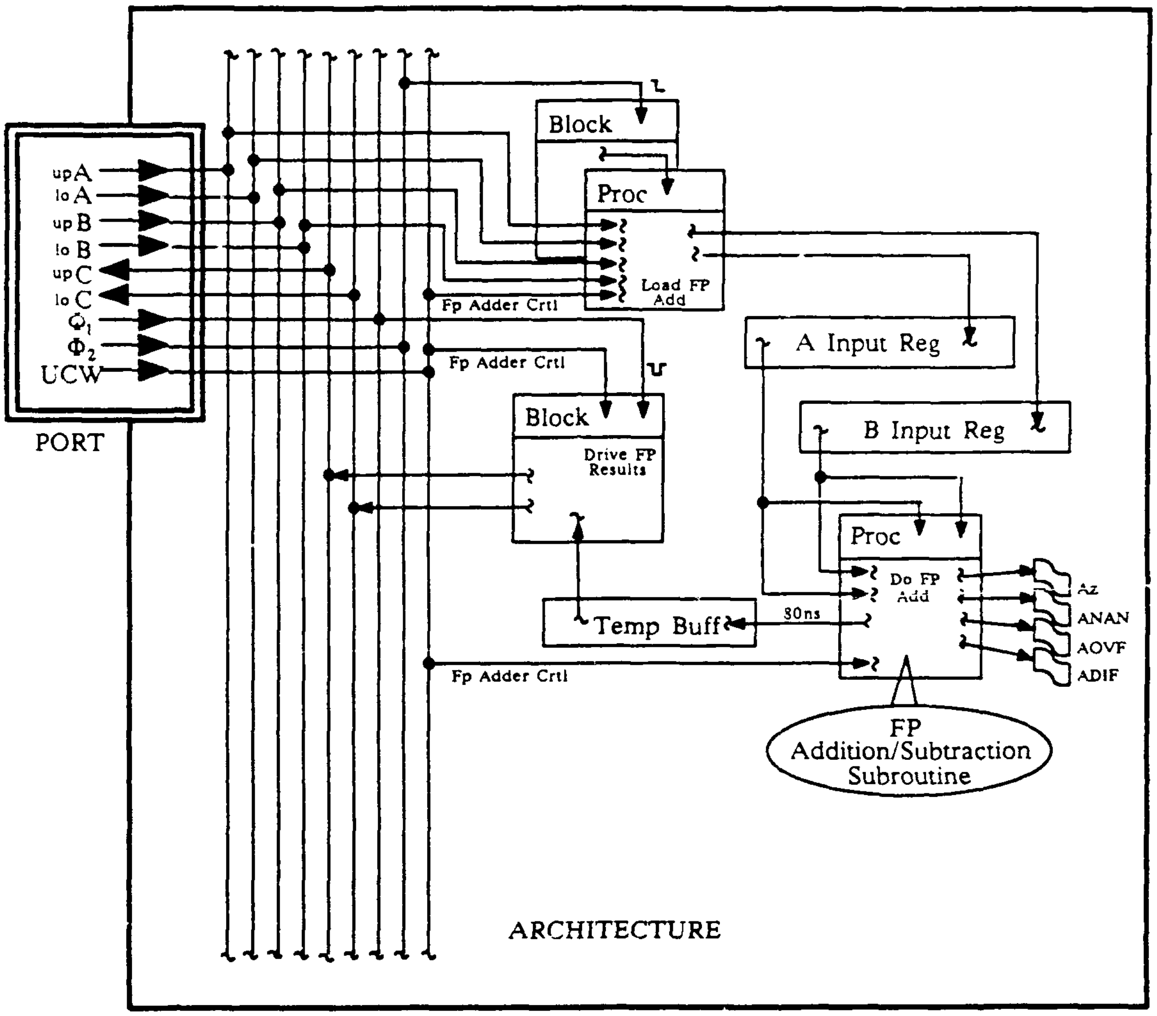


Figure 5.9. Floating Point Adder Entity Structure Diagram

The Floating Point Adder has a block/process statement combination that loads the data from the input busses to the input registers, followed by a process that performs the addition, which is followed by a block/process combination that drives the sum onto the output busses.

The first block/process statement combination loads the adder's input registers from the upper and lower **A** busses, and the upper and lower **B** busses. The guard condition of the block statement is True with the falling edge of PHI2 and when the Floating Point Adder Control micro–field selects to load the input registers. The nested process statement assigns the contents on the busses to appropriate input registers. The process statement is sensitive to changes occurring in the input registers. Any change to one of the input registers enables the process which conditionally test the Floating Point Adder Control micro–field to determine if a floating point addition is requested. If an addition is requested, a subroutine is called which performs the actual addition. The values contained in the input registers are passed to the subroutine, which in turn passes back the sum and the values of any flags that were set as a result of the computation. The returned sum is driven into holding registers after an 80ns delay. The last block/process statement drives the sum onto the output busses. The guard condition of the block statement is True when PHI1 is zero and when the Floating Point Adder Control micro–field decodes to drive the data out. The process statement which is sensitive to the implicit signal GUARD drives the results held in the temporary holding registers onto the upper and lower **C** busses. A final block statement, whose guard condition is True with the falling edge of PHI2 and when the Floating Point Adder Control micro–field decodes to drive results out, drives the flag values for the previous computation into the output flags.

The process developed to perform floating point subtraction is identical to that for floating point addition. The only exception is that the sign flag of the number off the **B** busses is inverted before the addition.

5.5.12 Floating Point Multiplier Model: Like the Floating Point Adder, the Floating Point Multiplier is a single macro-cell constructed using combinational logic to perform multiplication of double precision IEEE floating point numbers. The Floating Point Multiplier is also capable of multiplying two 32 bit integers which could generate a product that is 64 bits wide. Multiplication of two floating point numbers can be performed in two clock cycles when the multiplication is performed repeatedly in tight loops. Otherwise, three clock cycles are required - the extra clock cycle required to load the input registers.

The port list for the entity declaration of the design entity developed to model the Floating Point Multiplier is shown below.

```
entity FP_MULT is
port (UA_BUS   : inout   BUS_TYPE bus;
      LA_BUS   : inout   BUS_TYPE bus;
      UB_BUS   : inout   BUS_TYPE bus;
      LB_BUS   : inout   BUS_TYPE bus;
      UC_BUS   : inout   BUS_TYPE bus;
      LC_BUS   : inout   BUS_TYPE bus;
      PHI1     : in      BIT;
      PHI2     : in      BIT;
      UCW      : in      UPPER_ROM_WORD);
end FP_MULT;
```

Shown in the port list are the upper and lower **A** busses, the upper and lower **B** busses, the upper and lower **C** busses, clock signals PHI1 and PHI2, and the upper ROM Control Word. The architecture developed to model the behavior of the Floating Point Multiplier is identical to the architecture developed for the Floating Point Adder and is given in Figure 5.10.

A block/process statement reads the data off the input busses and loads the multipliers input registers. A single process statement which is sensitive the changes to the input registers performs the multiplication by calling a subroutine which performs the actual multiplication. The product returned from the subroutine is placed into temporary holding registers after an 80ns delay, and the flag values returned from the subroutine are also placed in temporary registers. A second block/process statement drives the product from the temporary registers onto the upper and lower **C** busses.

The portion of the architecture that models the integer multiply function, also shown in Figure 5.10, is again identical to that for floating point multiplication. The only exception, other than the differences in the Floating Point Multiplier Control micro–field, is that a separate subroutine is used to perform the integer multiplication.

5.5.13 External Memory Modeling: Although not an integral part of the FPASP, the external memory the FPASP interfaces to must be modeled for the VHDL simulator to function.

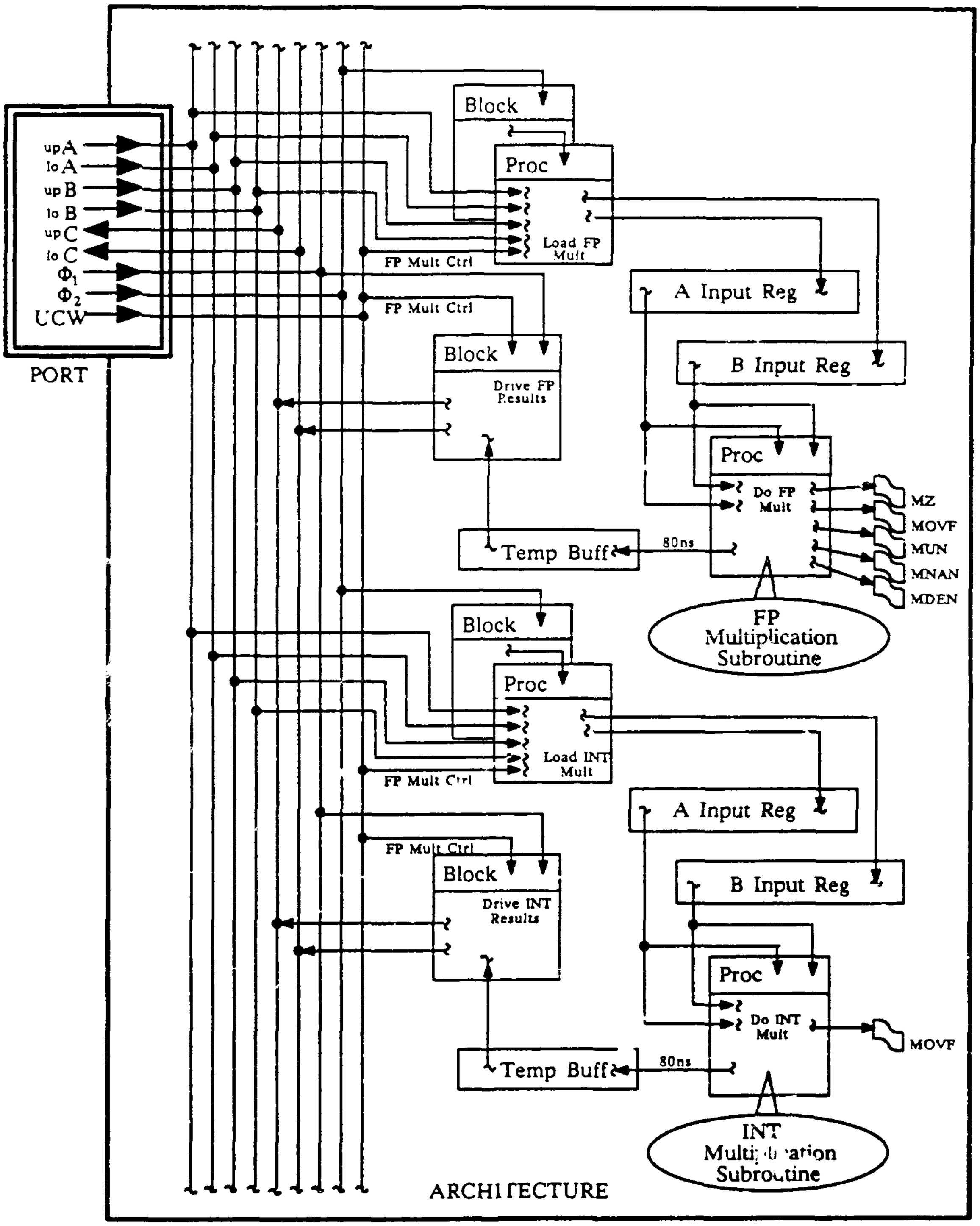


Figure 5.10. Floating Point Multiplier Entity Structure Diagram

The external memory configuration consists of two banks of the 32K × 32 bit words. One for the upper data path, and one for the lower data path. The port list for the entity declaration of the design entity declares the upper and lower address busses, the upper and lower **D** busses, both clock signals, and both the upper and lower ROM control words as shown.

```
entity ASP_MEMORY is
port (UD_BUS   : inout   BUS_TYPE bus;
      LD_BUS   : inout   BUS_TYPE bus;
      UP_ADD   : in      BUS_TYPE bus;
      LO_ADD   : in      BUS_TYPE bus;
      PHI1     : in      BIT;
      PHI2     : in      BIT;
      UCW      : in      UPPER_ROM_WORD;
      LCW      : in      LOWER_ROM_WORD);
end FP_MULT;
```

The architecture of the external memory design entity is shown in Figure 5.11. The architecture has two processes that read in the external memory files to initialize the memory arrays - one process for the even memory bank and the other for the odd memory bank. The processes then convert the bit-strings on the address busses to integers, and if a memory write is not requested, the memory arrays are indexed into and the memory array values are placed in temporary registers. If a memory write is requested, the values on the **D** busses are written into the memory arrays. These two processes also monitor the Done flag. When the Done flag is raised, the processes dump the contents of their respective memory banks to external disk files.

Placement of the memory contents stored in temporary registers onto the **D** busses is done by two block/process statements. These block/process statements monitor the Memory Write Enable Bar and the Memory Output Enable Bar micro–fields of both the upper and lower ROM control words. If a memory write is *not* requested, the contents of the temporary registers are driven onto the **D** busses.

5.5.14 Top-Level Design Unit: The top-level design unit of the VHDL simulator ties all the individual architectures together into an executable model. The top-level design unit has no port list in its declaration section. All of the busses, clock signals, and ROM control words are

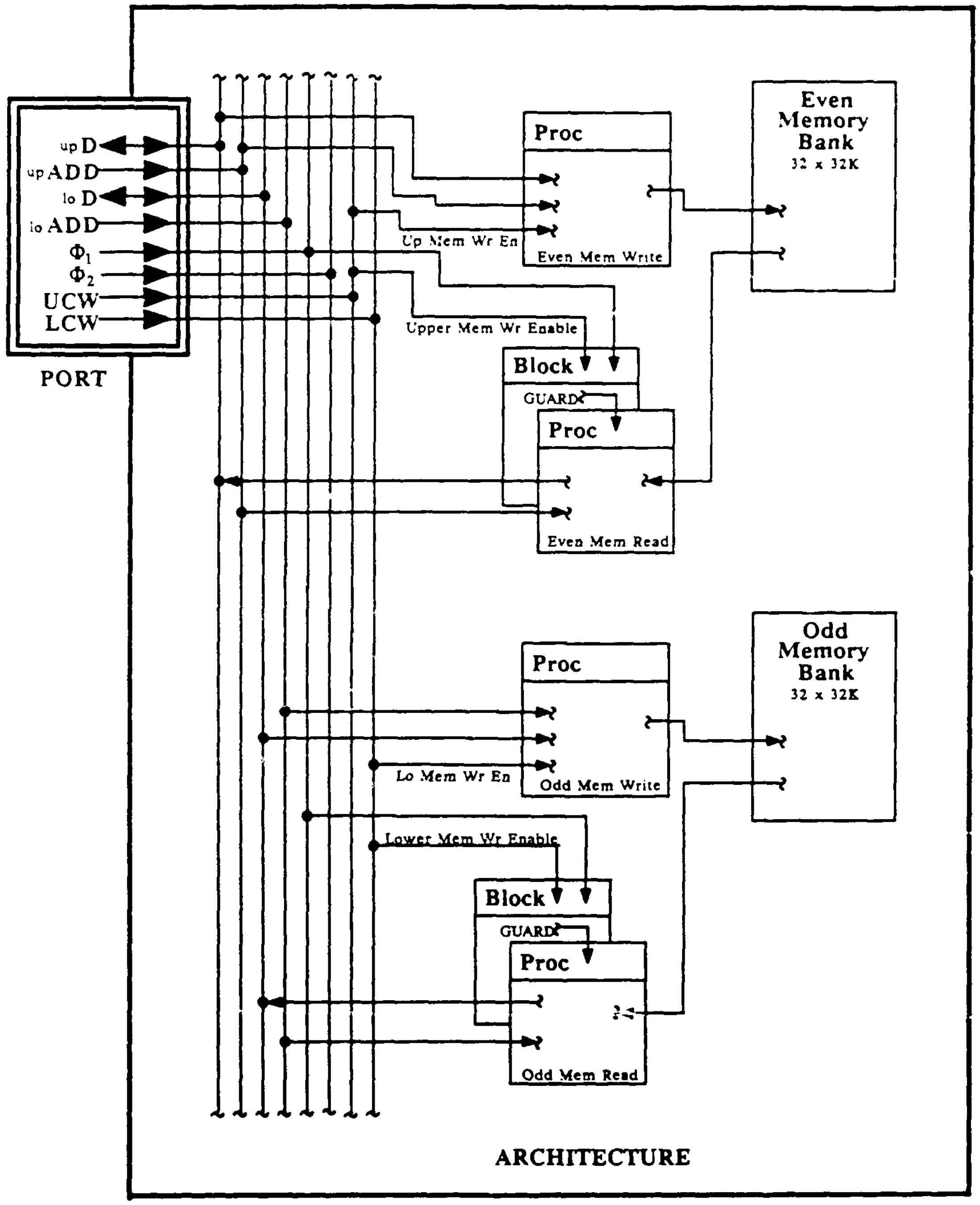


Figure 5.11. External Memory Entity Structure Diagram

declared in the architecture of the unit, and all the architectures modeling the components of the FPASP are instantiated in unit. The top-level design unit can be viewed as a back-plane that all the components are plugged into.

5.6 *VHDL Simulator Testing*

The same test plan that was developed for the logic simulator was used with the VHDL simulator. The small microcode test programs that tested the transferring of data between registers, the functions of the ALUs and Linear Shifters, the Floating Point Adder and Multiplier, the Literal Inserter, Bus Ties, and external memory were used to test the VHDL simulator. The matrix multiplication and Newton Raphson Inversion microcode programs were also executed with the VHDL simulator to test the conditional branching and subroutine call functions of the microsequencer. As stated in the requirements, the FPASP VHDL simulator had to generate the same output files as the logic simulator. The FPASP state file shows the register contents and the values on the bus during each simulation cycle, and the memory files allow an examination of the external memory contents after the simulation has terminated. These files are the key to determining if the VHDL simulator correctly executed a microcode program. However, they are only gross indicators of how well the VHDL simulator matches the designed timing requirements of the FPASP. To observe the timing aspects of the VHDL simulator, it is necessary to use the Report Generator provided in the VHDL support environment.

The Report Generator allows a user to look at signal transactions selectively. The user generates a Report Control Language text file listing the signals to be observed, and the Report Generator compiles a listing showing each transaction that occurred to the signal as well as the simulated time that it occurred.

The test plan used for the logic simulator was modified to include the development of Report Control Language files for each of the microcode test programs. The microcode programs were simulated and the FPASP state file and the memory files were examined to determine if the code was executed correctly by the simulator. The Report Generator was invoked next using the Report Control Language file for that particular microcode program. The listing generated by the Report Generator was examined to observe timing of the signal transactions. The design of the FPASP could then be said to be validated if the all the microcode programs executed correctly and the timing of the selected signal transactions occurred within the limitations set forth by the designer.

VI. Kalman Filter Microcode Development

6.1 Introduction

The rapid prototyping of the FPASP begins with the identification of an application which could best be served by the FPASP. Such applications are computationally intensive, can be specified algorithmically, and do not have special hardware requirements beyond those provided by the FPASP. The application chosen for this thesis is the Kalman Filter algorithm. The Kalman Filter algorithm is characterized by the large number of matrix algebra computations required to arrive at a solution. The benefits expected by implementing the Kalman Filter algorithm with the FPASP are a decrease in execution times without a degradation computational precision.

This chapter will look at the Kalman Filter from the viewpoint of the algorithm structure and not the theory. The data structures required by one Filter uses are identified, and the computations that must be performed are examined in detail. The chapter will also provide an introduction to programming with the FPASP microcode as the translation of the Kalman Filter algorithm is explained.

6.2 The Kalman Filter Algorithm

The Kalman Filter can be used in any application that cannot be observed directly to provide an inference of hidden values of interest from external measurements. The Kalman Filter can be defined as an *optimal recursive data processing algorithm* that estimates the state of a system indirectly (May79). The optimal nature of the Filter is beyond the scope of this thesis, however, the recursive description implies that the Filter does not require all previous data to be kept in storage and reprocessed every time a new measurement is taken. The fa[illegible] the Filter is a data processing algorithm implies the Filter is just a computer program.

The Kalman Filter is dependent on a number of externally initialized matrices to provide input data. Intermediate result matrices are calculated which are used by two routines, propagate and update, to compute the estimates of the system state. The propagation routine can be called independent of the update routine. However, before an update can be performed, a propagation must be performed.

6.2.1 Initial Data Structures: The Kalman Filter is slowed in the U–D factorized form in double precision. It performs a number of computations on matrix and vector data structures to arrive at an estimate of the state of a process. Many of these matrices and vectors must be initialized prior to invoking the Filter. The matrices and vectors that must be initialized are: the n-by-n *system dynamics matrix*, $\mathbf{F}(t_{i-1})$ (n is the number of state variables); the n-by-s *noise distribution matrix*, $\mathbf{G}(t_i)$ (s is the number of process noises); the s-by- s *driving noise strength matrix*, $\mathbf{Q}(t_i)$; the m-by-1 *measurement noise strength matrix*, $\mathbf{R}(t_i)$ (m is the number of measurements); the n-by-p *discrete control input distribution matrix*, $\mathbf{B}_d(t_{i-1})$ (p is the number of controls); the p-by-1 *controls input matrix*, $\bar{\mathrm{u}}(t_{i-1})$; the m-by-1 *measurement matrix*, $\bar{\mathbf{z}}$; the m-by-n *state observation matrix*, $\mathbf{H}(t_i)$; the n-by-1 *state vector*, $\hat{\mathrm{x}}(t_i^-)$; the initial upper triangular factorization of the *state covariance matrix*, $\mathrm{U}(t_i^-)$, and the initial diagonal matrix of the *state covariance matrix*, $\mathrm{D}(t_i^-)$.

The data structures identified above must be initialized by the host system prior to commanding the FPASP to begin its computations. Initializing the matrices requires that the host system access the external memory of the FPASP and write the matrix elements into memory in *row–major* form, one matrix after the other, in a predetermined order. The order the matrices must be loaded will be discussed later.

6.2.2 Computed Data Structures: The Kalman Filter algorithm performs several intermediate computations in the propagation routine. Computed are the n- by-n *state transition matrix*, $\boldsymbol{\Phi}(t_i, t_{i-1})$, and the n-by-n *discretized covariance matrix of process noises*, $\mathbf{Q}_d(t_{i-1})$, which is factorized into the upper triangular matrix $\mathbf{G}_T(t_i)$, and the diagonal matrix $\mathbf{Q}_D(t_i)$. The update routine computes the *Kalman gain matrix*, $\mathbf{K}(t_i)$ which is used to update the measurements into the measurement matrix z.

6.2.3 Propagation Calculations: The propagation routine propagates the state value estimates and the covariance matrix forward to the next sample time. The routine has been broken into five steps.

The first step of the propagation routine calculates the *state transition matrix*, $\boldsymbol{\Phi}(t_i, t_{i-1})$, using the series approximation

$$\boldsymbol{\Phi}(t_i, t_{i-1}) = \mathbf{I} + \sum_{k=1}^{x} \frac{1}{k!}\mathbf{F}^k(t_{i-1})\Delta t^k$$

where **I** is the identity matrix. Provisions are made for multiple propagation steps to account for time variations in $\mathbf{F}(t)$. This computation is performed for each propagation to propagate forward any changes in the **F** matrix that may have occurred in the interval Δt. The number of summations, x, performed depends on somewhat on the size of Δt, and how accurate the approximation needs to be. This variable x, as well as the Δt, are user selectable and needs to be set before performing the computation. Provisions are made for negative Δt, should the arrival time of a measurement require it.

The second step uses the approximated Φ matrix calculated in the first step along with the matrices **G** and **Q** to calculate the first order approximation of $\mathbf{Q}_d(t_{i-1})$ as follows

$$\mathbf{Q}_d(t_{i-1}) = \frac{\Delta t}{2}[\boldsymbol{\Phi}(t_i, t_{i-1})\mathbf{G}\ \mathbf{Q}\ \mathbf{G}^T + (\boldsymbol{\Phi}(t_i, t_{i-1})\mathbf{G}\ \mathbf{Q}\ \mathbf{G}^T)^T]$$

This step propagates any changes in the $\boldsymbol{\Phi}$ matrix which resulted from changes in the **F** matrix in the interval Δt.

The propagation of the state values is performed in the third step as follows

$$\bar{\mathbf{x}}(t_i) = \boldsymbol{\Phi}(t_i, t_{i-1})\bar{\mathbf{x}}(t_{i-1}) + \mathbf{B}_d(t_{i-1})\bar{\mathbf{u}}(t_{i-1})$$

The *discretized covariance matrix of process noises*, $\mathbf{Q}_d(t_{i-1})$, is factorized into the upper triangular matrix $\mathbf{G}_d(t_i)$, and the diagonal matrix $\mathbf{Q}_d(t_i)$ in the fourth step. Substituting matrix names to clear up subscripting

$$\mathbf{L} = \mathbf{Q}_d(t_{i-1}) \quad \mathbf{M} = \mathbf{Q}_D(t_i) \quad \mathbf{N} = \mathbf{G}_T(t_i)$$

for the last (n^{th} column) let

$$\mathbf{M}_{nn} = \mathbf{L}_{nn}$$

$$\mathbf{N}_{in} = \begin{cases} 1 & \text{if } i = n \\ \mathbf{L}_{in}/\mathbf{M}_{nn} & i = n-1, \ldots, 1 \end{cases}$$

for the other columns, $j = n-1, n-2, \ldots, 1$:

$$\mathbf{M}_{jj} = \mathbf{L}_{jj} - \sum_{k=j+1}^{n} \mathbf{M}_{kk} \mathbf{N}_{jk}^2$$

$$\mathbf{N}_{ij} = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \left(\mathbf{L}_{ij} - \sum_{k=j+1}^{n} \mathbf{M}_{kk}\mathbf{N}_{ik}\mathbf{N}_{jk}\right)/\mathbf{M}_{jj} & \text{otherwise} \end{cases}$$

This step completes computing the data structures required for the propagation routine.

Step five performs the **UD** propagation and contains the bulk of the routine. A temporary matrix, **T**, is formed taking the product of the $\boldsymbol{\Phi}$ and **U** matrices.

$$\mathbf{T} = \boldsymbol{\Phi}(t_{i+1}, t_i)\mathbf{U}(t_i^+)$$

The $\mathbf{G}_T(t_i)$ matrix calculated in step four is appended to the **T**

$$\mathbf{Y}(t_{i+1}^-) = [\,\mathbf{T} \ \vdots \ \mathbf{G}_T(t_i)\,]$$

forming an $n \times 2n$ **Y** matrix. The $\mathbf{D}(t_i^+)$ matrix is extended diagonally with the $\mathbf{Q}_D(t_i)$ matrix

$$\tilde{\mathbf{D}}(t_{i+1}^-) = \begin{bmatrix} \mathbf{D}(t_i^+) & \vdots & 0 \\ \cdots & & \cdots \\ 0 & \vdots & \mathbf{Q}_D(t_i) \end{bmatrix}$$

generating a $2n \times 2n$ matrix.

Viewing $\mathbf{Y}^T(t_{i+1}^-)$ as n column vectors of length $2n$

$$\mathbf{Y}^T(t_{i+1}^-) = [a_1 \ \vdots \ a_2 \ \vdots \ \ldots \ \vdots \ a_n]$$

the routine then iterates on the following relations for $k = n, n-1, \ldots, 1$:

$$\begin{aligned} c_{kj} &= \tilde{D}_{jj}(t_{i+1}^{-})a_{kj} \qquad j = 1, 2, \ldots, 2n \\ \mathbf{D}_{kk}(t_{i+1}^{-}) &= a_k^T c_k \\ d_k &= c_k / \mathbf{D}_{kk}(t_{i+1}^{-}) \end{aligned}$$

$$\left.\begin{aligned} \mathbf{U}_{jk}(t_{i+1}^{-}) &= a_j^T d_k \\ a_j &\leftarrow a_j - \mathbf{U}_{jk}(t_{i+1}^{-})a_j \end{aligned}\right\} j = 1, 2, \ldots, k-1$$

where $\leftarrow$ denotes the replacement, or writing over old variables. On the last iteration, for $k = 1$, only the first two relations need be computed.

6.2.4 Update Calculations: The update routine incorporates external measurements into the filter estimates. At time t_i, $\mathbf{U}(t_i^-)$ and $\mathbf{D}(t_i^-)$ are available from the previous time propagation. Using the measurement value z_i, the i^{th} column vector of $\mathbf{H}$, and the scaler 1–by–1 matrix $\mathbf{R}$

$$\begin{aligned} f &= \mathbf{U}^T(t_i^-)\mathbf{H}^T(t_i) \\ v_j &= \mathbf{D}_{jj}(t_i^-)f_j \quad j = 1, 2, \ldots, n \\ a_0 &= \mathbf{R} \end{aligned}$$

Then, for $k = 1, 2, \ldots, n$, calculate

$$\begin{aligned} a_k &= a_{k-1} + f_k v_k \\ \mathbf{D}_{kk}(t_i^+) &= \mathbf{D}_{kk}(t_i^-)a_{k-1}/a_k \\ b_k &\leftarrow v_k \\ p_k &= -f_k/a_{k-1} \end{aligned}$$

$$\left.\begin{array}{lcl} U_{jk}(t_i^+) & = & U_{jk}(t_i^-) + b_j p_k \\ b_j & \leftarrow & b_j + U_{jk}(t_i^-) v_k \end{array}\right\} j = 1, 2, \ldots, (k-1)$$

After n iterations, $U(t_i^+)$ and $\mathbf{D}(t_i^+)$ have been computed, and the filter gain $\mathbf{K}(t_i)$ can be calculated in terms of the n-vector $\mathbf{b}$ made up of the components $b_1, b_2, \ldots, b_n$ computed above. The state is then updated as

$$\begin{array}{lcl} \mathbf{K}(t_i) & = & \mathbf{b}/a_n \\ \hat{\mathbf{x}}(t_i^+) & = & \hat{\mathbf{x}}(t_i^-) + \mathbf{K}(t_i)\left[z_i - \mathbf{H}(t_i)\hat{\mathbf{x}}(t_i^-)\right] \end{array}$$

6.3 Microcode Development

It was noted in the previous sections that the Kalman Filter algorithm can be broken down to a series of initial data structures, computed data structures, and various calculations. The primary requirement for the developed microcode is to accurately and quickly implement the Kalman Filter. The various calculations highlight the large number of matrix algebra operations that need to be performed implying the microcode must efficiently perform these type of computations. Aside from being quick, the microcode must also efficiently use the external memory. The recursive nature of the Kalman Filter means that not all of the computations must be performed every time. This necessarily means that these intermediate results must be maintained in memory. This is in addition to the memory requirements for the initial matrices. The following sections will discuss the memory requirements of the Kalman Filter, the microcode subroutines developed to support the Filter calculations, and the Filter microcode development.

6.3.1 Memory Considerations: The Kalman Filter microcode must be efficient with its use of external memory. The performance of the FPASP will be seriously degraded if the host system must reinitialize the Filter data structures prior to every propagation or update. For optimal utilization of the memory, the microcode must allow for as many of the initial data structures to remain intact as possible, and still provide enough storage capacity to support computations. At most, it is expected that the FPASP must be able to compute a 64–state Kalman Filter. For this size Filter, the number of states n is 64, the number of measurements m is 32, the number of process noises s is 32, and the number of controls p is 16. The total initial memory requirements including reserved memory locations is 14,594 words. With a total external memory capacity expected to be 32,768 words, this leaves 18,174 words for intermediate computation and storage. The initial FPASP memory map is shown in Figure 6.1.

The first 32 words are reserved for error diagnostics and an instruction word at location U0. The 'U' in the memory index indicates the upper memory bank. The instruction word at location U0 is used to indicate to the Kalman Filter microcode whether a propagation is to be performed, or an update. The number of controls, p, is placed in U32, and the number of process noises, s, is placed in L32. The number of series expansions to be used in calculating Φ, x, is placed in U33, and the number of state variables, n, is placed in L33. The number of measurements, m, is placed in U34, and a pointer to the beginning of the $U(t_i^-)$ is placed in L34. Memory locations UL35 through UL40 are unused. The current time stamp, t_i, which is supplied by the host system, is placed in memory location UL41. The current time stamp is a double precision IEEE floating point number. The most significant 32 bits of the number are placed in U41, and the least significant 32 bits are placed in L41. The previous time stamp, t_{i-1} is placed in UL42, and like the current time stamp, it too is a double precision IEEE floating point number. The previous time stamp needs to be initialized to 0.00 only once prior to the first time propagation. This value will then be maintained by the microcode program. Each of the initial data structures are stored in row–major form with the first element of the first data structure stored at memory location UL50. Each data structure is then placed one after the other until all are in memory. The rest of the external memory is available for the computed data structures and as scratch memory for computations.

With the initial data structures mapped into external memory, it is now possible to begin the development of the Kalman Filter microcode. The initial mapping was necessary because the FPASP has no facilities for indirect addressing, and the primitive GMAT assembler has no provisions for associating a label with a memory location. Therefore, it is necessary to hardcode each memory location that must be written to or read from.

6.3.2 Microcode Subroutines: The Kalman Filter microcode program makes use of several microcode subroutines during the course of its computations. The most used microcode subroutine

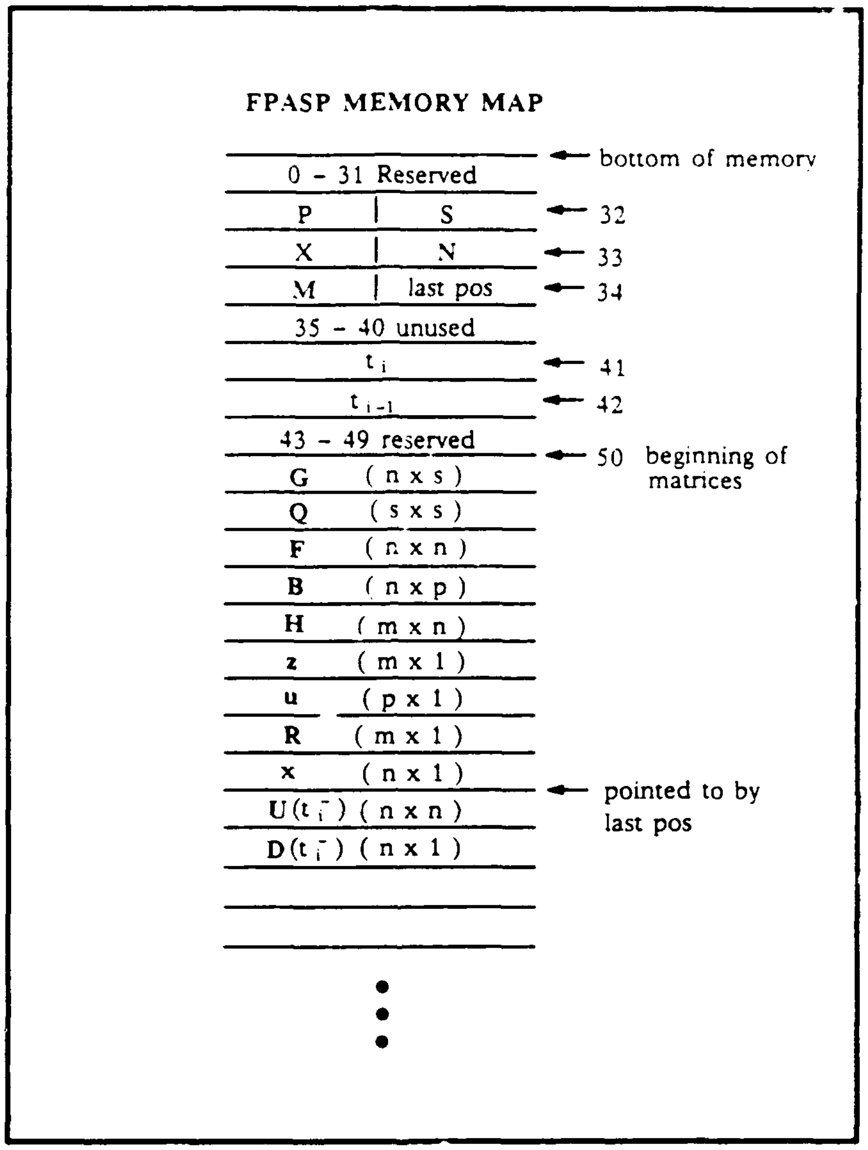


Figure 6.1. Initial Memory Map

is the Matrix Multiply routine developed by Capt John Comtois as part of his thesis (Com88). The basic operation in matrix multiplication is the dot product of two vectors where the elements of each vector are multiplied together and accumulated into a final result. For two vectors of size n, there are n multiplies and $(n - 1)$ additions required to compute the dot product. One of the primary objectives of implementing the Kalman Filter with FPASP is to realize a significant decrease in executions times. It is anticipated that it will require approximately three million floating point operations to compute a 64–state Kalman Filter, with a large percentage of the operations being performed in matrix multiplication. A matrix multiply routine that efficiently uses the FPASP floating point hardware is essential to ensuring the Kalman Filter microcode program meets the requirement of increased execution time. Before looking at the matrix multiply subroutine developed by Capt Comtois, it would help to look at the operation of the floating point hardware, and how the hardware can efficiently be used in performing operations on data arrays.

It was previously noted that latency of the floating point hardware is 2 clock cycles if the floating point operation is being performed in a loop where the adder or multiplier can be loaded and read in the same clock cycle. Otherwise, a latency of three clock cycles is required because of the extra cycle required to load the adder or multiplier. The following example illustrates how a floating point operation is placed in a loop to provide a latency of 2 clock cycles.

```
1)   SAMPLE:                                BPT+            MAR=EU    ETIE    E=BPT
                                                            MAR=EL;
      ;
2)                                                  R1=DU
                                                    R1=DL;
     ;
3)   AU=R1     BU=R25               FP*L
     AL=R1     BL=R25;
     ;
4)                                  FP*     BPT+    UIN1+   MAR=EU    ETIE    E=BPT
                                                            MAR=EL;
     ;
5)                                                  R1=DU
                                                    R1=DL;
     ;
     LOOP:
6)   AU=R1     BU=R25    MBR=CU     FP*LD   APT+    R1=DU   MAR=EU    ETIE    E=APT    UIN1N     BR
     AL=R1     BL=R25    MBR=CL                     R1=DL   MAR=EL                          LOOP;
     ;
7)                                          BPT+    UIN1+   MAR=EU    ETIE    E=BPT    WEBU
                                                            MAR=EL                     WEBL;
```

In this example, a vector is stored in external memory, and the program will scaler multiply the vector and store the results to a second memory location. The first code line (each code line consists of two mnemonic lines; one for the upper ROM and one for the lower ROM) loads both the upper and lower MARs with an address to external memory from the B memory address pointer.

The B memory address pointer is simultaneously incremented to the next address location. The effect of incrementing the address will not be sensed at the output of the address pointer until the next clock cycle. The second code line reads in the contents at the external memory location into the upper and lower general purpose registers R1. The third code line loads the floating point multiplier with the memory value from the R1s and a previously stored scalar value from the R25s. The next line of code performs the multiplication and reloads the MARs with the address to the next element of the vector. Also, the upper incrementable register, UIN1, is being used to count the number of iterations of the loop. Here UIN1 is incremented to reflect the fact that one element has been read in and scalar multiplied. The fifth line of code is necessary to read in the next element of the vector prior to entering the loop. The main loop begins in the sixth line of code. The next vector element and the scalar value is loaded into the floating point multiplier at the same time the previous product is being driven out into the upper and lower MBRs. The address for the result is placed into the MARs from the A memory pointer, and the next element of the vector is read into the R1s. UIN1 is checked to see if it has incremented to zero. If it hasn't, a conditional branch will be made to the label LOOP. Because of the pipeline architecture of the FPASP, the next line of code after a conditional branch is always executed prior to the branch. Making use of this, the seventh line of code writes the result from the MBRs out to memory, reloads the MARs with the address to the next element of the input vector, and increments UIN1. The program continues in this fashion until UIN1 increments to zero, and the program falls through the loop.

The example program above illustrates the number of preparatory steps required to set up the registers, counters, and pointers prior to entering the loop. At first, this may seem an inefficient way to code the scalar multiplication of a vector. In the true sense of software design, this may be true, however, the intent is program in a way to use the FPASP hardware in the most efficient way possible to minimize the number of clock cycles required for a particular operation. In this program, it will require $5 + 2(n - 1)$ clock cycles to scalar multiply a vector with n elements. If n is large, the 5 clock cycles required to prepare the registers becomes insignificant.

6.3.2.1 Matrix Multiply: The matrix multiply routine developed by Capt Comtois is a double–nested loop, with a dot product routine in the inner loop. For square matrices with n vectors, the dot product routine must be called n^2 times with each dot product performing n

multiplies and $(n-1)$ additions. Thus, n^3 multiplies and $n^2(n-1)$ additions must be done for each matrix multiplication. The inner loop, or the dot product routine is shown below and it performs not only floating point multiplication, but also floating point addition in the same loop.

```
1)  DOTP:
                         R25=CU   GNDCU               R1=DU   MAR=EU   ETIE   E=APT
                         R25=CL   GNDCL               R1=DL   MAR=EL;
    ;
2)  AU=R25   BU=R25               FP+L               MB=DU   MAR=EU   ETIE   E=BPT
    AL=R25   BL=R25                                  MB=DL   MAR=EL;
    ;
3)  AU=MBR   BU=R1               FP*L    FP+   APT+  R1=DU   MAR=EU   ETIE   E=APT
    AL=MBR   BL=R1                             LIN1+ R1=DL   MAR=EL;
    ;
    DPLOOP:
4)  AU=R25   BU=FP+              FP*   FP+L    BPT+  MB=DU   MAR=EU   ETIE   E=BPT   LIN1N   BR
    AL=R25   BL=FP+                                  MB=DL   MAR=EL                  DPLOOP;
    ;
5)  AU=MBR   BU=R1    R25=CU   FP*LD   FP+   APT+   R1=DU   MAR=EU   ETIE   E=APT
    AL=MBR   BL=R1    R25=CL               LIN1+  R1=DL   MAR=EL;
    ;
6)  AU=R25   BU=FP+            FP+L
    AL=R25   BL=FP+;
    ;
7)  AU=R22                     PASSU   FP+                   MAR=EU   ETIE
                               PASSL                         MAR=EL;
    ;
8)                    R25=CU         FP+D                                    TRU    RET
                      R25=CL;
    ;
9)  AU=R25            MBR=CU   MOVUN   PASSU
    AL=R25            MBR=CL   MOVLN   PASSL;
```

The first three lines of code in the dot product subroutine prepare registers and the floating point hardware prior to entering the loop that performs the multiplications and accumulates the products. The first line of code clears the R25s to zero which will accumulate the products, and reads in the first element from one of the vectors from memory. The second code line clears the floating point adder by loading it with zero, and reads in the first element from the other vector from memory. The third line of code loads the floating point multiplier with the first elements from both vectors, reads in the second element from the first vector from memory, and increments the loop counter. The actual dot product loop begins with the fourth line of code. On initial entry into the loop, the floating point adder is loaded with the contents of the R25s and the output of the floating point adder. Since both have been initialized to zero, the result will be zero also. Still in the fourth line of code, the next element of the second vector is read in from memory, and an conditional branch back through the loop is selected should the loop counter not be equal to zero. In the fifth line of code, the next two elements are loaded into the floating point multiplier, the next element of the first vector is read in from memory, and the loop counter is incremented. The fourth and fifth lines of code are repeated until the loop counter reaches zero. Each time through the loop, the contents of the R25s are added to the current sum in the floating point adder. Exiting the loop, the last product to be accumulated is loaded into the floating point adder from the R25s in the sixth line of code, added in the seventh line of code, and driven into the R25s for a last time in the eighth line of code. The subroutine is complete and an unconditional return is made to the calling routine after placing the dot product into the MBRs.

The outside loop (this portion of the subroutine can be reviewed in Appendix B) of the matrix multiply routine is responsible for initializing the registers and pointers to each succeeding vector in the matrix. The total number of clock cycles required by the matrix multiply subroutine to multiply a m-by-n times n-by-p is

$$\text{Matrix Multiply Clock Cycles} = 2pmn + 11pm + 2p - 1$$

Multiplying two 64–by–64 matrices together using the matrix multiply subroutine will require 569,471 clock cycles to perform the required 266,176 floating point operations. A useful metric for examining the efficiency of a microcode routine is the ratio of floating point operations to the number of clock cycles. For any given routine, the optimal execution time is equal to the total number of floating point operations required times the period of the clock. Therefore, the ratio of the floating point operations to clock cycles indicates the overall utilization of the floating point hardware and how close the routine is to being optimal. Taking the ratio of floating point operations to clock cycles for the matrix multiply routine yields a 0.4674 utilization factor for the floating point hardware. Had the code been designed inefficiently with three lines of code in the inner dot product loop, the number of clock cycles required to multiply the same two 64–by–64 matrices would have jumped to 831,615 dropping the utilization factor to 0.32.

6.3.2.2 Matrix Addition: The next subroutine developed to support the Kalman Filter microcode program is the matrix addition subroutine. As with the matrix multiply subroutine, the code was designed to make efficient use of the floating point hardware by making the addition loop as tight as possible. The significant difference between the multiply and addition subroutines, is that the addition routine requires two memory reads and one memory write each time through the loop. Since one complete clock cycle is required for each memory access, the optimum number of code lines within the loop is three. The addition subroutine is a double–nested loop in which the inner loop performs vector addition. The outer loop sets up the registers and memory pointers for each successive vector in the two matrices. The number of clock cycles required to add two m-by-n matrices is

$$Matrix\ Addition\ Clock\ Cycles = 4nm + 5m$$

which, for two 64–by–64 matrices, equates to 16,704 clock cycles to perform the necessary 4096 floating point additions. The utilization factor for the floating point hardware is 0.245

In retrospect, the matrix addition subroutine does not use the floating point hardware as efficiently as it could. The main fault is that there are four code lines in the inner addition loop where the optimum number could be three. The development of this subroutine occurred early in the learning curve of FPASP microcode development. A more efficient means to add to matrices is to utilize the fact that the matrices are stored in row–major form and treat both matrices as extended vectors. Doing so, there would be a need for only one loop rather than the double–nested loop currently implemented. Perhaps in a later version of the Kalman Filter microcode program this avenue will be explored, however, the gain in increased hardware utilization will not be that significant. At best, the number of clock cycles required for matrix addition is $3mn$ for two m-by-n matrices. For two 64–by–64 matrices this equates to 12,288 clock cycles yielding a 0.3333 utilization factor for the floating point adder. This is only a 8.83% increase over the current implementation.

6.3.2.3 Matrix Scalar Multiplication: A third subroutine developed to support the Kalman Filter microcode program is the Matrix Scalar Multiplication subroutine. This is an extremely efficient routine that makes use of the fact that matrices are stored in row–major form. The entire subroutine uses seven lines of microcode. The first four microcode lines initialize the registers and memory pointers, and load the floating point multiplier with the first matrix element. The next two microcode lines form the multiplication loop that reads in successive matrix elements, performs the scalar multiplication, and writes the products out to memory. The entire subroutine requires only $2mn+4$ clock cycles, which for a 64–by–64 matrix equates to 8,196 clock cycles. The optimum number of clock cycles for scalar multiplication is $2mn$ which for the same size matrix is 8,192 clock cycles. The efficiency of this subroutine is 99.95%.

6.3.2.4 Newton–Raphson Inversion: The fourth and last subroutine used by the Kalman Filter microcode program is the elegant adaptation of the Newton–Raphson method developed by Capt John Comtois (Com88). The FPASP does not have a floating point divider, and the reciprocal of a floating point number is required in several of the Kalman Filter computations. The Newton–Raphson method approximates the reciprocal of a number using only floating point multiplication and floating point addition (Cav84). The Newton–Raphson method

$$x_{i+1} = x_i(2 - Bx_i)$$

iterates recursively until

$$\lim_{i \to \infty} x_i = \frac{1}{B}$$

and

$$\lim_{i \to \infty} Bx_i = 1$$

The objective is to develop a quotient with as few iterations as possible, and with the least number of steps per iteration. A ROM used to contain a table of initial values for x_0 that can be indexed by the high–order bits of the divisor can greatly reduce the initial error thereby reducing the number of iterations.

Shown below is the microcode developed by Capt Comtois based on this scheme of storing initial values in the FPASP Function ROM.

```
     NRI:
 1)            BU=R25  R23=CU    NANDU    PASSU                   IMZU
                       R23=CL             GNDCL                             #1111111111110000;
     ;
 2)            BU=R23  R23=CU    ADDU     PASSU                   IMZU
                                                                            #0111111111110000;
     ;
 3)            BU=R23  R24=CU    RCP
                       R24=CL             GNCL;
     ;
 4)  AU=R23    BU=R24  R23=CU    ORU      PASSU                   IMZU
                                                                            #1111111111111100;
     ;
 5)                    R24=CU             PASSU                   IMZU
                                                                            #0100000000000000;
     ;
 6)  AU=R25    BU=R23                              FP*L                     TRU        BR
     AL=R25    BL=R23                                                                  RLP2;
     ;
 7)                    IN1=CL                      FP*    ILNL
                                 MOVLN    PASSL                             #1111111111111101;
     ;
     RLP:
 8)  AU=R25    BU=FP*                              FP*LD
     AL=R25    BL=FP*;
     ;
 9)                                                FP*                      TRPS       CALL
                                                                                       TRAP;
     ;
     RLP2:
10)  AU=R24     BU=FP*                             FP*D   FP-L
     AL=R24     BL=FP*;
     ;
11)                                                       FP-               TRPS       CALL
                                                                                       TRAP;
     ;
12)  AU=R23    BU=FP+                              FP*L
     AL=R23    BL=FP+;
     ;
13)                                                FP*                      TRPS       CALL
                                                                 LIN1+                 TRAP;
     ;
14)                                                                         LIN1N      BR
                                                                                       RLP;
     ;
15)                      R23=CU                    FP*D
                         R23=CL;
     ;
16)                                                                             TRU  RET;
```

In Capt Comtois' routine, the number to be inverted, or divisor, is used to generate a seed for selecting the initial value for x_0 from the Function ROM. The divisor is in the R25s, and the quotient is returned in the R23s. To generate the seed, the first line of microcode uses the Literal Inserter to insert a mask on the upper A bus which isolates the sign bit and the exponent bits of the divisor. The sign and exponent bits are then inverted by the NAND operation of the upper ALU. The second line of microcode adds 1022 to the exponent. The result is -e-1 in IEEE format. The sign bit gets re–inverted back to its original value as a result of the carry out of the exponent addition. This seed is then placed on the upper B bus in the third line of microcode to index into the Function ROM and the partial initial value for x_0 is place in UR24. In the fourth line of microcode, partial initial value from the Function ROM is added to the seed value used to index into the Function ROM, and the bits in the resultant mantissa are ANDed off by the value inserted on the upper A bus by the Literal Inserter. The resultant value is the initial quess, or the initial value for x_0. This method of determining the initial value for x_0 insures that only four iterations are required to arrive at a quotient. Carefully tracing through the rest of the routine shows that Newton–Raphson's iterative method is begin performed as given in the above equation. For any given divisor, the number of clock cycles required by this routine to arrive at a quotient is constant at 31 clock cycles. The number of floating point operations performed is also constant at 12 yielding an utilization factor of 0.387 for the subroutine. However, at 25MHz, it takes only 1.26μsecs to calculate the inverse of a number, and a few more percentage points one way or the other will be of little consequence in the overall timing requirements of the program.

6.3.3 Propagation Routine: The propagation routine was derived directly from the propagation calculations presented in section 6.2.3. The routine begins by computing $\Phi(t_i, t_{i-1})$ which is then followed by the propagation of the state values in $x(t_i)$. The routine then computes $Q_d(t_{i-1})$, and then factorizes $Q_d(t_{i-1})$ into $Q_d(t_i)$ and $G_d(t_i)$. Finally, the routine performs the UD propagation and returns to the calling higher–level routine. This section will examine each phase of the propagation routine.

6.3.3.1 Computing $\Phi(t_i, t_{i-1})$: The computation of $\Phi(t_i, t_{i-1})$ is performed first in the propagation routine. The algorithm developed to calculate $\Phi(t_i, t_{i-1})$ is shown below in a pseudocode format as it is implemented in microcode.

```
$\Delta t = t_i - t_{i-1}$
for $j = 1$ to $n^2$ do
    $T_1[j] = F[j] \times \Delta t_1$          ;treat F and $T_1$ as extended vectors
    $T_2[j] = F[j]$
end for
for $j = 1$ to $n$ do
    $T_1[j][j] = T_1[j][j] + 1.0$
end for
```

```
if $x = 1$ skip the rest
for $i = 2$ to $x$ do
    SCALAR = $\Delta t^i/i!$
    for $j = 1$ to $n$ do
        for $k = 1$ to $n$ do
            $T_3[k][j] = 0$
            for $l = 1$ to $n$ do
                $T_3[k][j] = T_2[k][l] \times F[l][j]$
            end for
            $T_1[k][j] = T_1[k][j] + (T_3[k][j] \times SCALAR)$
        end for
    end for
    for $j = 1$ to $n^2$ do
        $T_2[j] = T_3[j]$
    end for
end for
```

This portion of the propagation routine approximates $\Phi(t_i, t_{i-1})$ by performing a truncated matrix exponential (May79). The code begins by subtracting the previous time stamp from the current time stamp to determine Δt for the propagation. Treating the **F** matrix as an extended vector, the routine then scalar multiplies **F** by Δt in a single loop placing the results in the temporary matrix $\mathbf{T}_1$. In the same loop, a copy of **F** is placed in the temporary matrix $\mathbf{T}_2$. In the actual microcode, this is accomplished setting a second pointer to point to the **F** matrix. The identity matrix, **I**, is then added to the $\mathbf{T}_1$ matrix. This is performed by simply adding one to each diagonal element. Next x, the number of series expansions, is tested. If x is one, $\Phi(t_i, t_{i-1})$ is computed and a pointer is set to point to the beginning of $\mathbf{T}_1$. Otherwise, a loop is entered that for $i = 2$ to x, performs the remaining iterations of the series expansion. Once inside the loop, the microcode calculates i factorial. This is performed using two register sets. Both register sets are initialized to 1.00. Register set two is incremented by one each time through the loop, and the value in register set one is multiplied by the value in register set two each time through the loop generating the appropriate factorial. The Newton–Raphson Inversion subroutine is used for each iteration to calculate the inverse of the factorial. Raising Δt to the appropriate power is done is a similar fashion. One register set which was initialized to Δt is multiplied by Δt each time through the loop. A scalar value of $\Delta t^i/i!$ is then computed. Two n-by-n scratch areas of memory, $\mathbf{T}_2$ and

$\mathbf{T}_3$, are required to compute the powers of the **F** matrix. To perform the remaining computations in the series expansion requires a matrix multiplication, followed be a scalar/matrix multiplication, and finally a matrix addition. This could have been performed by making unconditional calls to the appropriate subroutines. This would however incur unnecessary overhead. To overcome this, the matrix multiply routine was modified to perform all three operations at once. After each dot product, the result is multiplied by the scalar value computed above, and added to the appropriate element in the $\mathbf{T}_1$ matrix. The updating of $\mathbf{T}_2$ with the new power of **F** is depicted in the pseudocode as transferring the element values from $\mathbf{T}_3$ to $\mathbf{T}_2$ in a loop. The microcode performs this operation by swapping pointers to the beginning of the matrices. When this portion of the routine is complete, $\mathbf{T}_1$ is $\Phi(t_i, t_{i-1})$ and a pointer is set to the beginning of the matrix that can be referenced later.

The number of clock cycles required to compute $\Phi(t_i, t_{i-1})$ is dependent on x. Counting up everything, it will require $2n^2+2n+39+(2n^3+19n^2+50)(x-1)$ clock cycles. The number of floating point operations required are also dependent on x and are $n^2+n+1+(2n^3+n^2+16)(x-1)$. For a 64-by-64 matrix and $x = 1$, 8359 clock cycles are required, and 4161 floating point operations yielding a hardware utilization factor of 0.497. For the same matrix size and $x = 2$, 610521 clock cycles are required, and 532561 floating point operations, which yields a hardware utilization factor of 0.872.

6.3.3.2 Propagating State Values: After computing $\Phi(t_i, t_{i-1})$, the state values in $\bar{x}(t_{i-1})$ are propagated. An algorithm derived from the arithmetic calculations was developed to propagate the state values and is shown below.

```
for $i = 1$ to 1 do
    for $j = 1$ to $n$ do
        $T_1[j][i] = 0$
        for $k = 1$ to $n$ do
            $T_1[j][i] = \Phi(t_i, t_{i-1})[j][k] \times \bar{x}(t_i)[k][i]$
        end for
    end for
end for
if M[0] bit 31 = 1 then
```

```
    for $i = 1$ to 1 do
        for $j = 1$ to $n$ do
            $T_1[j][i] = 0$
            for $k = 1$ to $p$ do
                $T_2[j][i] = B_d(t_{i-1})[j][k] \times \bar{u}(t_i)[k][i]$
            end for
        end for
    end for
    for $i = 1$ to $n$ do
        $\bar{x}(t_i)[i] = T_1[i][1] + T_2[i][1]$
    end for
end if
```

This portion of the propagation routine begins by calculating the matrix product of $\Phi(t_i, t_{i-1})$ and $\bar{x}(t_{i-1})$. To perform this operation, the microcode initializes the necessary registers and pointers with the array sizes and the beginning addresses to the matrices, and then calls the matrix multiply subroutine. The results of the matrix multiplication are placed in a temporary scratch area of memory. After returning from the matrix multiplication subroutine, the microcode then tests the most significant bit of the word in external memory at location zero. If this bit is set, the controls inputs are computed and added to the previous result. The matrix multiplication subroutine is used to compute the product of the two matrices, $B_d(t_{i-1})$ and $\bar{u}(t_{i-1})$. The matrix addition subroutine is then used to add the two results together. The final result, $\bar{x}(t_i)$, is in the temporary scratch area of memory. $\bar{x}(t_i)$ is moved from the temporary scratch area, and is overwritten into the memory area containing $\bar{x}(t_{i-1})$ at the completion of the propagation routine.

The number of clock cycles required to propagate the state values is $2n^2 + 13n + 22$ if the MSB of M[0] is 0. Otherwise, it requires $2n^2 + 2np + 13(n + p) + 29$ clock cycles. The number of floating point operations are easily calculated and are $2n^2 - n$ for the case where the MSB of M[0] is '0', and $2n^2 + 2np - 2n$ when the MSB of M[0] is '1'. For dimension sizes of n equal to 64, and p equal to 32, 9046 clock cycles, and 8128 floating point operations are required for the case when the MSB of M[0] is '0'. In the case when the MSB of M[0] is '1', 13536 clock cycles, and 12160 floating point operations are required. The first case realizes a hardware utilization factor of 0.898, and the second case realizes a hardware utilization factor of 0.898 also.

6.3.3.3 Computing $\mathbf{Q}_d(t_{i-1})$*:* After propagating the state values in $\hat{x}(t_{i-1})$, the propagation routine computes $\mathbf{Q}_d((t_{i-1})$. The algorithm derived from the propagation calculations is shown below.

```
for $i = 1$ to $s$ do
    for $j = 1$ to $n$ do
        $\mathbf{T}_1[j][i] = 0$
        for $k = 1$ to $n$ do
            $\mathbf{T}_1[j][i] = \mathbf{\Phi}(t_i, t_{i-1})[j][k] \times \mathbf{G}[k][i]$
        end for
    end for
end for
for $i = 1$ to $s$ do
    for $j = 1$ to $n$ do
        $\mathbf{T}_2[j][i] = 0$
        for $k = 1$ to $s$ do
            $\mathbf{T}_2[j][i] = \mathbf{T}_1[j][k] \times \mathbf{Q}[k][i]$
        end for
    end for
end for
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $\mathbf{T}_1[j][i] = 0$
        for $k = 1$ to $s$ do
            $\mathbf{T}_1[j][i] = \mathbf{T}_2[j][k] \times \mathbf{G}[i][k]$
        end for
    end for
end for
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $\mathbf{T}_2[i][j] = \mathbf{T}_1[i][j] + \mathbf{T}_1[j][i]$
        end for
    end for
end for
for $i = 1$ to $n^2$ do
    $\mathbf{Q}_d(t_{i-1}[i] = \mathbf{T}_2[i] \times \Delta t/2$
end for
```

This portion of the microcode begins by multiplying $\Phi(t_i, t_{i-1})$ and G together and placing the results in the temporary matrix $\mathbf{T}_1$. $\mathbf{T}_1$ is then multiplied by $\mathbf{Q}$, and the results are placed in temporary matrix $\mathbf{T}_2$. $\mathbf{T}_2$ is then multiplied by $\mathbf{G}^T$ and the results placed in $\mathbf{T}_1$. The matrix multiply subroutine is general enough to allow the multiplication of one matrix by the transpose of another matrix. This is accomplished by adjusting the distance between the rows of the second matrix to equal the distance between the columns, and the distance between the columns is set to equal the distance between the rows. The temporary matrix $\mathbf{T}_1$ now has the results of the computation $\Phi(t_i, t_{i-1})\mathbf{GQG}^T$. The next portion of the code adds $\Phi(t_i, t_{i-1})\mathbf{GQG}^T$ to its transpose. The matrix addition subroutine is also general enough to allow one matrix to be added to the transpose of another matrix. This is accomplished in the same fashion as the matrix multiplication subroutine. After performing the matrix addition, the microcode multiplies the matrix sum by $\Delta t/2$ to complete the computation of $\mathbf{Q}_d(t_{i-1})$.

The number of clock cycles required to compute $\mathbf{Q}_d(t_{i-1})$ is $4n^2s + 26ns + 2ns^2 + 19n^2 + 10(n+s) + 42$. The number of floating point operations performed is $4n^2s + 2ns^2 - 2ns + n^2 + 12$. For dimension sizes of $n = 64$ and $s = 32$, 787434 clock cycles, and 589836 floating point operations are required for the computation. The hardware utilization factor for this portion of the microcode is 0.8323.

6.3.3.4 $\mathbf{Q}_d(t_{i-1})$ *Factorization:* The propagation routine after computing $\mathbf{Q}_d(t_{i-1})$, moves on to factorize $\mathbf{Q}_d(t_{i-1})$ into the upper triangular matrix $\mathbf{G}_d(t_i)$, and the diagonal matrix $\mathbf{Q}_d(t_i)$. The pseudocode algorithm for the factorization of $\mathbf{Q}_d(t_{i-1})$ is given below.

```
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        if $i = j$ then
            $\mathbf{G}_d(t_i)[i][j] = 1$
        else
            $\mathbf{G}_d(t_i)[i][j] = 0$
        end if
            $\mathbf{Q}_d(t_i)[i][j] = 0$
    end for
end for
```

```
for $j = n$ downto 1 do
    $\mathbf{Q}_d(t_i)[j][j] = \mathbf{Q}_d(t_i, t_{i-1})[j][j]$
    if $(\mathbf{Q}_d(t_i) = 0$ then
        $AA = 0$
    else
        $AA = 1/\mathbf{Q}_d(t_i)[j][j]$
    end if
    for $k = 1$ to $j - 1$ do
        $BB = \mathbf{Q}_d(t_i, t_{i-1})$
        $\mathbf{G}_d(t_i) = AA \times BB$
        for $i = 1$ to $k$ do
            $\mathbf{Q}_d(t_i, t_{i-1})[i][k] = \mathbf{Q}_d(t_i, t_{i-1})[i][k] - (BB \times \mathbf{G}_d(t_i)[i][j])$
        end for
    end for
end for
$\mathbf{Q}_d(t_i)[1][1] = \mathbf{Q}_d(t_{i-1})[1][1]$
```

The microcode begins the factorization of $\mathbf{Q}_d(t_{i-1})$ by initializing $\mathbf{G}_d(t_i)$ to a n-by-n identity matrix, and initializing $\mathbf{Q}_d(t_i)$ to zero. After initializing the matrices, the microcode enters a loop that transfers the j^{th} diagonal element of $\mathbf{Q}_d(t_{i-1})$ to the j^{th} diagonal element position of $\mathbf{Q}_d(t_i)$. The temporary variable, AA, is then assigned either 0 or the inverse of the j^{th} diagonal element of $\mathbf{Q}_d(t_{i-1})$. The first nested loop traverses the j^{th} column of $\mathbf{Q}_d(t_{i-1})$ assigning the k^{th} element to the temporary variable BB. The k^{th} element of the j^{th} column of the matrix $\mathbf{G}_d(t_i)$ is then assigned the product $AA \times BB$. The second nested loop then modifies the elements of $\mathbf{Q}_d(t_{i-1})$. The result is the factored upper triangular matrix, $\mathbf{G}_d(t_i)$, and the diagonal matrix, $\mathbf{Q}_d(t_i)$, such that $\mathbf{Q}_d(t_{i-1}) = \mathbf{G}_d(t_i)\ \mathbf{Q}_d(t_i)\ \mathbf{G}_d^T(t_i)$.

The number of clock cycles required to perform the factorization of $\mathbf{Q}_d(t_{i-1})$ is $(4n^3 + 18n^2 + 110n - 66)/3$, and the number of floating point operations performed are $(2n^3 + 3n^2 + n - 6)/6$. For a 64−state Kalman Filter, it will require 376470 clock cycles, and 89439 floating point operations yielding a 0.2376 hardware utilization factor for this portion of the microcode. It is interesting to note the significant drop in efficiency. This can be attributed to the algorithm which, although it does deal with matrices, does not lend itself well to the architecture of the FPASP.

6.3.3.5 UD Propagation: The final portion of microcode in the Propagation routine performs the **UD** propagation. Except for the updating of the state values, all the microcode prior to this has been computing intermediate matrix values needed for the propagation of the $\mathbf{U}(t_i^-)$ and $\mathbf{D}(t_i^-)$ matrices. The following pseudocode highlights the UD propagation routine.

```
for $i = 1$ to $n$ do
    for $j = 1$ to $n + s$ do
        $\mathbf{Y}[i][j] = 0$
    end for
end for
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        for $k = 1$ to $n$ do
            $\mathbf{Y}[i][j] = \mathbf{Y}[i][j] + \left(\boldsymbol{\Phi}(t_i, t_{i-1})[i][k] \times \mathbf{U}(t_{i+1}^-)[k][j]\right)$
        end for
    end for
    for $l = 1$ to $n$ do
        $\mathbf{Y}[i][l + n] = \mathbf{G}_d(t_i)[i][l]$
    end for
end for
for $i = 0$ to $2n$ do
    $\tilde{\mathbf{D}}(t_{i-1}^-)[i] = 0$
end for
for $i = 1$ to $n$ do
    $\tilde{\mathbf{D}}(t_{i-1}^-)[i] = \mathbf{D}(t_i^-)[i]$
end for
for $i = 1$ to $n$ do
    $\tilde{\mathbf{D}}(t_{i-1}^-)[i] = \mathbf{Q}_d(t_i)[i][i]$
end for
for $k = n$ downto 1 do
    for $j = 1$ to $2n$ do
        $c[j] = \tilde{\mathbf{D}}(t_{i-1}^-)[j] \times \mathbf{Y}[k][j]$
    end for
    $\mathbf{D}(t_i^-)[k] = 0$
    for $j = 1$ to $2n$ do
        $\mathbf{D}(t_{i+1}^-)[k] = \mathbf{D}(t_{i+1}^-)[k] + (\mathbf{Y}[k][j] \times c[j])$
    end for
    for $j = 1$ to $2n$ do
        $d[j] = c[j]/\mathbf{D}(t_{i+1}^-)[k]$
    end for
    for $i = 1$ to $k - 1$ do
        $\mathbf{U}(t_{i+1}^-)[i][k] = 0$
        for $j = 1$ to $2n$ do
```

```
            $U(t_{i+1}^-)[i][k] = U(t_{i+1}^-)[i][k] + (Y[i][j] \times d[j])$
         end for
         for $j = 1$ to $2n$ do
            $Y[i][j] = Y[i][j] - \left(U(t_{i+1}^-)[i][k] \times Y[k][j]\right)$
         end for
      end for
   end for
```

The final section of microcode in the propagation routine begins by initializing $\mathbf{Y}(t_{i-1}^-)$ to zero. After initializing the matrix to zero, the microcode performs the matrix multiplication of $\boldsymbol{\Phi}(t_i, t_{i-1})$ and $\mathbf{U}(t_i^+)$ placing the resultant matrix directly in rows 1 through n, and columns 1 through n of $\mathbf{Y}(t_{i-1}^-)$ which is a n-by-$2n$ matrix. This is accomplished by adjusting the distance between the rows of the resultant matrix in the subroutine Matrix Multiplication. The microcode then copies $\mathbf{G}(t_i)$ into rows 1 through n, and columns $n+1$ through $2n$ of $\mathbf{Y}(t_{i-1}^-)$. Once the microcode has completed generating $\mathbf{Y}(t_{i-1}^-)$, the vector $\tilde{\mathbf{D}}(t_{i+1}^-)$ is initialized to zero, and $\mathbf{D}(t_i^+)$ is copied into the first n element positions. Next, the diagonal matrix, $\mathbf{Q}(t_i)$, which is stored as an n length vector, is copied into the remaining n element positions of $\tilde{\mathbf{D}}(t_{i+1}^-)$. The microcode then enters a loop which iterates from $k = n$ downto 1. The temporary vector, $\mathbf{c}$, is calculated for each iteration which is a point–by–point vector multiply of $\tilde{\mathbf{D}}(t_{i+1}^-)$ and the k^{th} row of $\mathbf{Y}(t_{i-1}^-)$. Next the dot product of the k^{th} row of $\mathbf{Y}(t_{i-1}^-)$ and $\mathbf{c}$ form the new k^{th} element of $\mathbf{D}(t_i^+)$. The temporary vector $\mathbf{d}$ is generated next which is formed by multiplying $\mathbf{c}$ by the inverse of the k^{th} element of $\mathbf{D}(t_i^+)$. The microcode then enters an inner loop where the k^{th} column of $\mathbf{U}(t_i^+)$ from rows 1 to k are propagated by performing the dot products of the i^{th} rows of $\mathbf{Y}(t_{i-1}^-)$ and $\mathbf{d}$. Finally, $\mathbf{Y}(t_{i-1}^-)$ is modified in preparation for the next iteration of the outer loop.

The number of clock cycles required by the microcode to perform the **UD** propagation is $13n^3 + 29n^2 + 63n - 6$, and the number of floating point operations required are $6n^3 + 3n^2 + 12n$. For a 64–state Kalman Filter where $n = 64$, this equates to 3,530,682 clock cycles, and 1,585,920 floating point operations which gives a hardware utilization factor of 0.4492.

6.3.3.6 Propagation Routine Summary: The Kalman Filter propagation microcode routine was designed and developed incrementally in conjunction with a higher–order language

FPASP Propagation Performance							
n	m	s	p	x	Cycles	FLOPS	ΔT
states	measurements	noises	controls	terms	$x\ 10^6$	$x\ 10^6$	msecs
64	32	32	16	1	4.712	2.343	188.4
64	32	32	16	2	5.314	2.903	212.5
40	20	20	0	1	1.196	0.576	47.8
40	20	20	0	2	1.354	0.642	54.2
20	6	6	0	1	0.155	0.065	6.2
20	6	6	0	2	0.178	0.073	7.1
16	1	6	0	1	0.086	0.035	3.4
16	1	6	0	2	0.099	0.040	3.9

Table 6.1. Kalman Filter Propagation Performance

model of the same routine. In this manner, it was possible to test the results of each subportion of the microcode against another model. Final testing though required using the propagation microcode routine on actual navigational data provided as a *truth* model. The method used and the results obtained in the final testing will be discussed later in the chapter.

Table 6.1 summarizes the performance of the Kalman Filter propagation microcode routine for varying dimensions.

The first case is a 64–state Kalman Filter with one term in the series expansion of $\Phi(t_i, t_{i-1})$. For this case, the propagation microcode demonstrates a floating point hardware utilization factor of 0.497, indicating that the FPASP operating at 25MHz is performing 12.43 million floating point operations per second. As the array sizes of the data structures decreases, the number of floating point operations per second drops off in the last case of a 16–state Kalman Filter with two terms in the series expansion of $\Phi(t_i, t_{i-1})$. In this case, the propagation microcode uses the floating point hardware 39.8% of the time, indicating that a 25MHz FPASP is performing only 9.95 million floating point operations per second. Clearly, the microcode overhead involved in setting up each of the individual portions of the routine becomes the dominate factor as the sizes of the data structures is decreased.

Execution of the propagation microcode routine alters the FPASP memory map as depicted in Figure 6.2. The initial data structures, except for the state vector and the UD factors of the state covariance matrix, have been unaltered. The previous time stamp, t_{i-1}, now contains the current time stamp, and t_i must be updated prior to the next propagation.

6.3.4 Update Routine: The update microcode routine incorporates external measurements into the filter estimates. The microcode itself is independent of the propagation routine microcode, and is called as an independent subroutine. The development of the update routine proceeded as did the propagation routine, and was derived directly from the calculations presented in section 6.2.4. The update routine needs not compute any intermediate data structures independent of the routine itself. The following pseudocode program shows the algorithm developed for the update microcode routine.

```
for $l = 1$ to $m$ do
    for $1 = 1$ to $n$ do
        $c[i] = 0$
        for $j = 0$ to $n$ do
            $c[i] = c[i] + \left(\mathbf{U}^T(t_i^-) \times \mathbf{H}^T(t_i)\right)$
        end for
    end for
    for $i = 1$ to $n$ do
        $\mathbf{v}[i] = \mathbf{D}(t_i^-)[i] \times \mathbf{c}[i]$
    end for
    $\mathbf{a}[0] = \mathbf{R}[l]$
    for $k = 1$ to $n$ do
        $\mathbf{a}[k] = \mathbf{a}[k-1] + (\mathbf{c}[k] \times \mathbf{v}[k])$
        $\mathbf{D}(t_i^+)[k] = \mathbf{D}(t_i^-)[k] \times (\mathbf{a}[k-1]/\mathbf{a}[k])$
        $\mathbf{d}[k] = \mathbf{v}[k]$
        if $k \rangle 1$ then
            $\mathbf{p}[k] = -\mathbf{c}[k]/\mathbf{a}[k-1]$
            for $j = 1$ to $k - 1$ uo
                $\mathbf{temp} = \mathbf{U}(t_i^-)[j][k]$
                $\mathbf{U}(t_i^+)[j][k] = \mathbf{U}(t_i^-)[j][k] + (\mathbf{d}[j] \times \mathbf{p}[k])$
                $\mathbf{d}[j] = \mathbf{d}[j] + (\mathbf{temp} \times \mathbf{v}[k])$
            end for
        end if
    end for
    for $j = 1$ to $n$ do
        $\mathbf{K}(t_i)[j] = \mathbf{d}[j]/\mathbf{a}[n]$
    end for
```

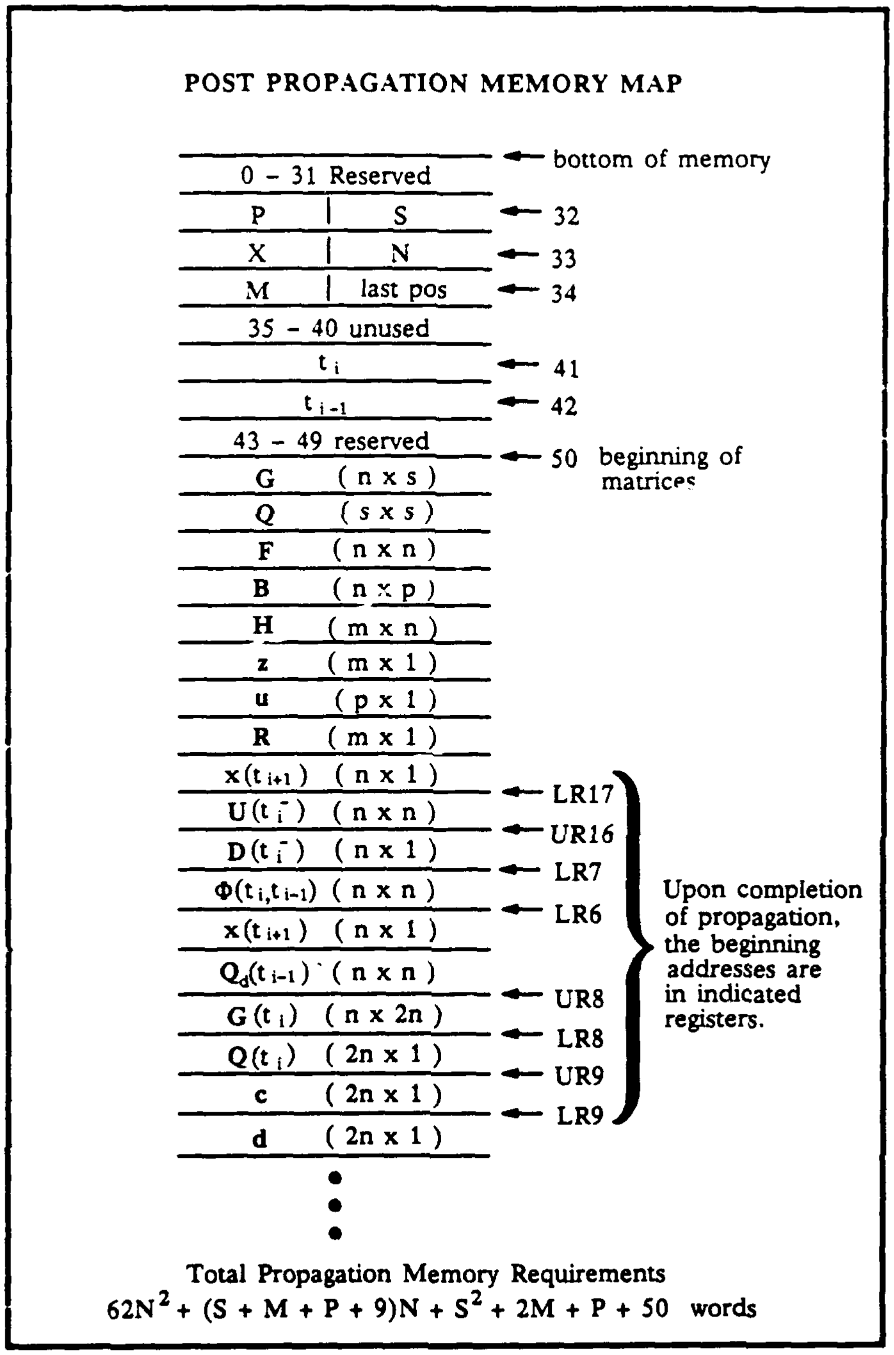


Figure 6.2. Post Propagation Memory Map

```
        temp = 0
        for $j = 0$ to $n$ do
            temp = temp + $\left(\mathbf{H}(t_i)[l][j] \times \bar{\mathbf{x}}(t_i^-)[j]\right)$
            temp = $\mathbf{z}[l]$ − temp
            $\bar{\mathbf{x}}(t_i^+)[j] = \bar{\mathbf{x}}(t_i^-)[j] + (\mathbf{K}(t_i)[j] \times \text{temp})$
        end for
    end for
```

The update microcode routine begins by entering a loop that iterates for each measurement. The temporary vector **c** is calculated by multiplying $\mathbf{U}^T(t_i^-)$ by the l^{th} row vector of $\mathbf{H}^T(t_i)$. Next, the temporary **v** vector is calculated by performing a point–by–point vector multiplication of $\mathbf{D}(t_i^-)$ and **c**. The next step is to assign the l^{th} measurement noise covariance value from **R** to the 0^{th} element of the temporary vector **a** which has a vector length of $n + 1$. The first nested loop is entered where the k^{th} element of **a** is calculated, the k^{th} element of $\mathbf{D}(t_i^+)$ is calculated, and the k^{th} element of the temporary vector **d** is assigned the k^{th} element value from **v**. If this is the first iteration, the rest of the loop is skipped, else for $k = 2$ to n, the temporary vector **p** is calculated and a second nested loop is entered. In the second nested loop, $\mathbf{U}(t_i^+)$ is updated, and the j^{th} element of **d** is modified. After completing both nested loops, the update routine calculates the filter gain, $\mathbf{K}(t_i)$, and the state values are updated.

The total number of clock cycles required for the FPASP to perform the update routine are $m(2n^2 + 121n + (13n(n-1)/2) + 47) + 13$. The total number of floating point operations performed are counted to be $m(4n^2 + 17n + 21)$. For a 64–state Kalman Filter with 32 measurements, it will require 1,219,181 clock cycles, and a total of 559,776 floating point operations. This yields a 0.459 hardware utilization factor for the update routine.

6.3.4.1 Update Summary: The same method that was used for developing the Kalman Filter propagation routine was used to develop the Kalman Filter update routine. The update routine was developed in conjunction with a higher–order language implementation of the same routine, which allowed incremental testing of the routine's subcomponents. Use of the update routine is in conjunction with the use of the propagation routine. Prior to a call to the update routine, the propagation routine must be called first to propagate to the Filter estimates to the

FPASP Propagation/Update Performance							
n	m	s	p	x	Cycles	FLOPS	ΔT
states	measurements	noises	controls	terms	$x\ 10^6$	$x\ 10^6$	msecs
64	32	32	16	1	6.062	2.902	242.5
64	32	32	16	2	6.664	3.169	266.6
40	20	20	0	1	1.561	0.718	62.4
40	20	20	0	2	1.719	0.784	68.8
20	6	6	0	1	0.189	0.076	7.6
20	6	6	0	2	0.213	0.085	8.5
16	1	6	0	1	0.090	0.036	3.6
16	1	6	0	2	0.103	0.041	4.1

Table 6.2. Kalman Filter Propagation/Update Performance

current time. Final testing of the update routine requires that the navigational data provided by a truth model be propagated first. Discussion of the final testing of the Kalman Filter microcode program is deferred until later in the chapter.

The expected performance of the Kalman Filter microcode program, which includes the performance of the update routine, is given in the following Table. The number of clock cycles and floating point operations, as well as the expected times reflect the operations of both the propagation routine and the update routine.

The first entry in the above Table indicates that four complete propagations and updates of a 64–state Kalman Filter with 32 measurements and one term in the series expansion of $\Phi(t_i, t_{i-1})$ can be performed every second. The overall floating point hardware utilization factor is 0.4973 indicating that the FPASP is performing on the average 12.43 million floating point operations per second. On the other hand, 244 propagations and updates can be performed with a 16–state Kalman Filter with one measurement and one term in the series expansion of $\Phi(t_i, t_{i-1})$. In this latter case, the floating point hardware efficiency has dropped to 40.11% and the average number of floating point operations per second has declined to 10.03 million. As it was indicated before, the microcode overhead for initializing and setting up the individual computations has become the dominate factor with a smaller Kalman Filter.

The execution of the update routine has altered the FPASP memory map as shown below

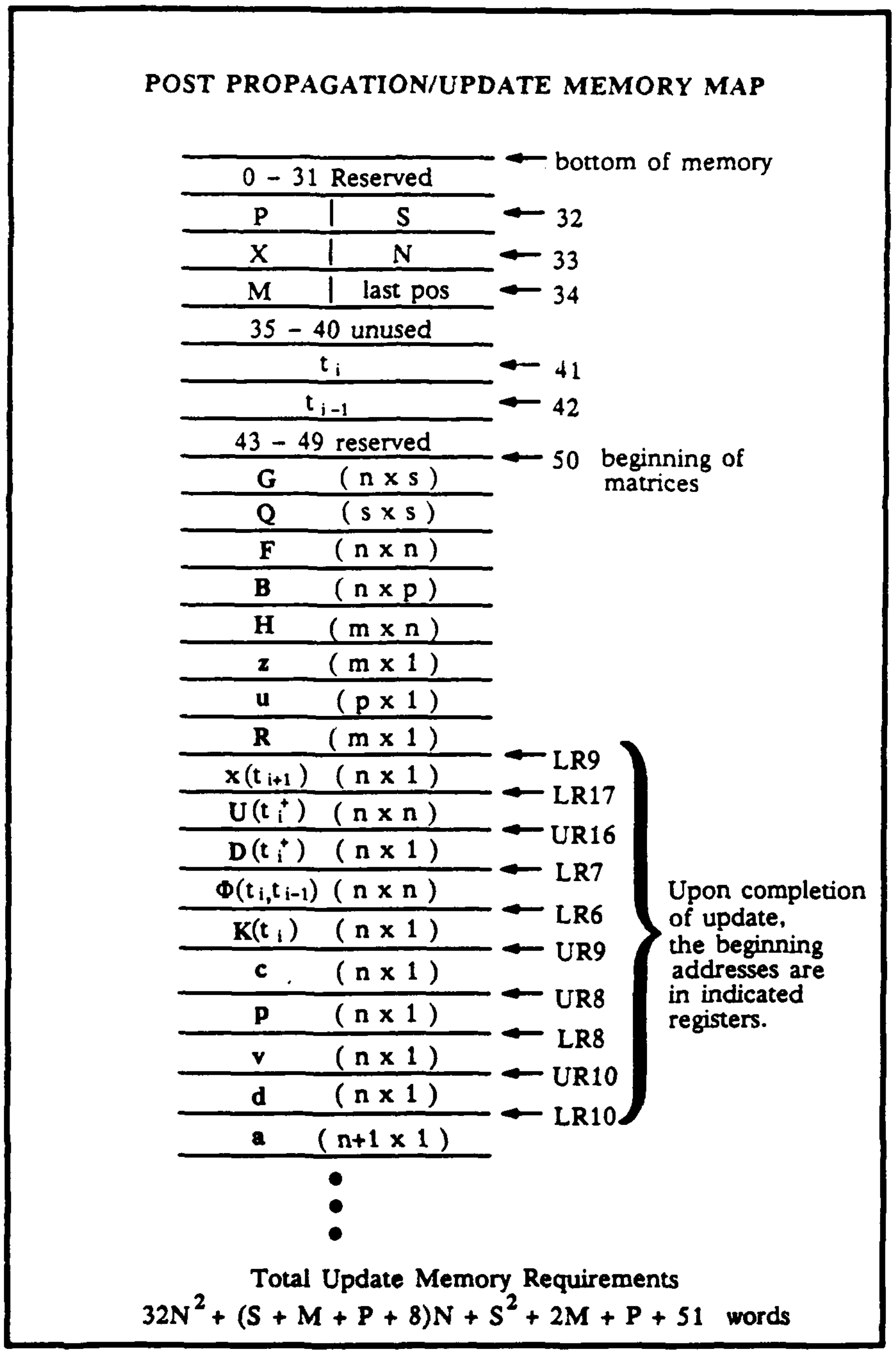


Figure 6.3. Post Propagation/Update Memory Map

in Figure 6.3. The initial data structures initialized by the host system have remained unaltered except for those that were updated. The host system needs update only individual array elements as necessary before the next propagation/update is performed.

6.4 Kalman Filter Microcode Program Testing

The previous sections went to some length to describe the algorithms developed to compute the calculations of the Kalman Filter. These algorithms were then implemented using FPASP microcode and the C programming language.

6.4.1 Kalman Filter Microcode Program Verification: As each phase of the microcode implementation was completed, verification of its operation and results were made against the output of the equivalent C routine. The verification of the C implementation was done against handed computed examples on very small models. The data structures were initialized using random data for the most part. Matrices that required being symmetric, positive semidefinite, were so constructed using random data values. The intent was to verify that the microcode routines manipulated the data correctly. To perform the validation of the Kalman Filter microcode program, it was necessary to obtain the data from an actual navigational model.

6.4.2 Kalman Filter Microcode Program Validation: The navigational model used to validate the Kalman Filter microcode program was a 4–state Kalman Filter used to track the location of a satellite in orbit around the earth (May82). The system dynamics matrix, **F**, was given as a function of the state vector $\bar{\mathbf{x}}$.

$$\mathbf{F}(t_{i-1}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ x_4^2 + \frac{2}{x_1^3} & 0 & 0 & 2x_1x_4 \\ 0 & 0 & 0 & 1 \\ \frac{2x_2x_4}{x_1^2} & \frac{-2x_4}{x_1} & 0 & \frac{-2x_2}{x_1} \end{bmatrix}$$

$$\hat{\mathbf{x}}(t_i^-) = \begin{bmatrix} 1 & 0 & 0 & 1.1547 \end{bmatrix}$$

The initial values of the remaining data structures were:

$$\mathbf{H}(t_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} 1.0\mathrm{E}-4 & 0 \\ 0 & 1.0\mathrm{E}-4 \end{bmatrix}$$

$$\mathbf{Q}(t_i) = \begin{bmatrix} 1.0\mathrm{E}-6 & 0 \\ 0 & 1.0\mathrm{E}-6 \end{bmatrix}$$

$$\mathbf{G}(t_i) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{P}(t_{i+1}^-) = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 0.25 \end{bmatrix}$$

The $\mathbf{P}(t_{i+1}^-)$ matrix is the state covariance matrix from which $\mathbf{U}(t_{i+1}^-)$ and $\mathbf{D}(t_{i+1}^-)$ are factorized. The measurement matrix, $\mathbf{z}$, was initialized with the initial measurements for the model.

$$\mathbf{z}(t_i^-) = \begin{bmatrix} 1.000017 & 1.1546876\mathrm{E}-2 \end{bmatrix}$$

These measurements are externally provided to the Filter, and must be updated by the host system prior to each update.

The validation of the Kalman Filter microcode program required that the logic simulator be modified to operate as a subroutine in order to perform successive propagations and updates. Normally, the logic simulator performs a single execution of a microcode program and then terminates. A higher–level validation routine was developed which first initialized the data structures to the values given above, then computed $\mathbf{F}(t_{i-1})$ based on the current state values $\hat{\mathbf{x}}(t_i^+)$, and then factorized $\mathbf{P}(t_{i+1}^-)$ into $\mathbf{U}(t_{i+1}^-)$ and $\mathbf{D}(t_{i+1}^-)$. The FPASP validation routine then called the FPASP logic simulator, which performed the time propagation first followed by a measurement update. The estimates of interest were the updated state values, $\bar{\mathbf{x}}(t_i^+)$, which estimated the position of the satellite in polar coordinates, and the state covariance matrix, $\mathbf{P}(t_{i-1}^+)$, which provides the variance of the estimates. After each propagation/update, the FPASP validation routine would update $\mathbf{z}$, and reperform the propagation/update.

The truth model for the FPASP Kalman Filter was developed by Capt F. Britt Snodgrass. Capt Snodgrass developed the truth model program using the above initial values, and executed the program using MSOFE (Multimode Simulation for Optimal Filter Evaluation) for one second of simulation time with Δt set to 0.01 seconds. The same data was then run using the modified FPASP logic simulator, and the results compared. Figure 6.4 gives the percent difference

$$\text{Percent Difference} = \frac{\left|\bar{\mathbf{x}}_i(t_i^+)_{MSOFE} - \bar{\mathbf{x}}_i(t_i^+)_{FPASP}\right|}{\bar{\mathbf{x}}_i(t_i^+)_{MSOFE}} \times 100\%$$

between the elements of the state vector of the truth model and the elements of the state vector of the FPASP model under evaluation.

Figure 6.4 shows that the difference between the elements of the state value vector of the MSOFE model, and the elements of the state value vector of the FPASP model is less than one percent.

The diagonal elements of the $\mathbf{P}(t_i^+)$ matrix were also compared. Figure 6.5 gives the percent difference of the diagonal elements, and shows that the diagonal elements of the covariance matrix differ by less than one percent.

The results of the validation run show that the Kalman Filter microcode program results are tracking the results of the truth model to within one percent after one second of simulation time. The differences experienced can be explained by noting that MSOFE propagates the state value estimates using Kutta Merson integration, whereas the Kalman Filter microcode routine propagates the state value estimates using an approximated state transition matrix. A longer run might reveal that the results are in fact diverging from one another, or that they are reaching some steady state error. In either case, the results after one second are a good match, and the conclusion can be

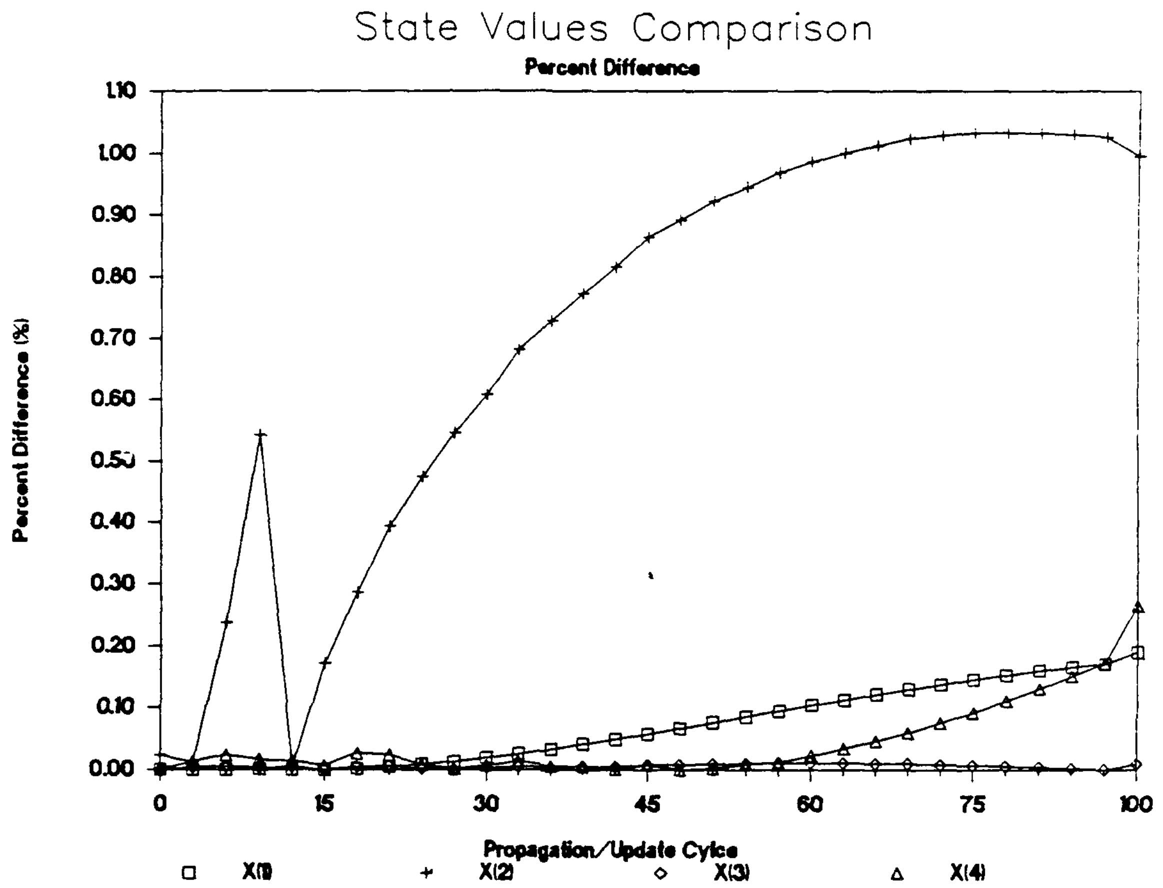


Figure 6.4. State Values Comparison

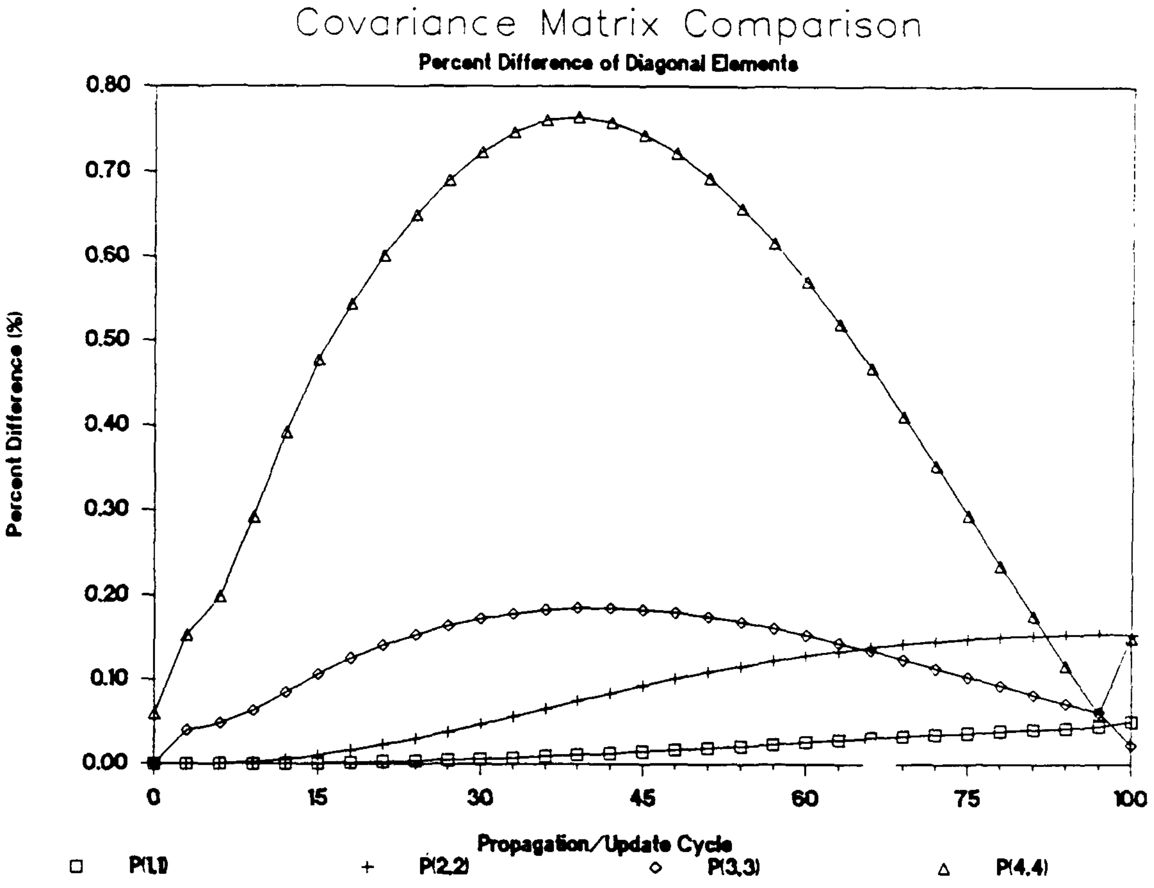


Figure 6.5. Covariance Matrix Comparison

made that the Kalman Filter microcode program does in fact perform the Kalman Filter algorithm hence validating the program (Sno89).

Comparing the results between the Kalman Filter microcode program and the validation model reveals that the developed microcode program is closely tracking the validation model. The conclusion that can be drawn is that the microcode program has accurately implemented the Kalman Filter algorithm, and hence the Kalman Filter microcode program is validated.

VII. Conclusions and Recommendations

7.1 Conclusions

This th sis has described the development of the Kalman Filter microcode which will be programmed into the FPASP, the development and use of a RTL simulator written in the C programming language, and the development of a register transfer level VHDL model of the FPASP. The development of the RTL simulator was deemed necessary when it was demonstrated that using the register transfer level VHDL model, VHDL simulator, for microcode development would require very long simulation times.

The development of the Kalman Filter microcode demonstrates more than the FPASP's capability of implementing a complex algorithm. It also demonstrates the versatility of the FPASP highly parallel micro-instruction set. The Kalman Filter application required 482 lines of microcode which includes all subroutines. This leaves 302 lines of microcode available out of the total 784 (Koc88). Many of the operations performed in the Kalman Filter algorithm are matrix algebra computations. The development of the microcode for these operations was simplified by the optimized hardware design of the FPASP. The remaining operations performed in the Kalman Filter algorithm can be best characterized as irregular. The most predominant irregular feature are triple nested loops that scan through one or more matrix data structures performing operations on each element within the matrices. For these operations, the best method found for developing the microcode was to lay out the basic operations required (multiplication, addition, subtraction, etc.) in a sequential fashion, and then more or less backfill using unused micro-fields to initialize registers and perform memory I/O. Although this may not be the most optimal approach, the net result was a significant decrease in projected execution times as compared to the execution times derived in the EE588 Kalman Filter class project.

Tablereftbl:Microcode Performance Comparisons compares the projected execution times of the Kalman Filter microcode developed in this research against that developed in the EE588 class project. The Table shows that the execution times for a 64-state Kalman Filter with one term in the series expansion of $\Phi(t_i, t_{i-1})$ decreased by approximately 51%.

Again considering the first case in Table 7.1, an optimal execution time would be the total number of floating point operations required by the Kalman Filter algorithm. From Table 6.2, the

Microcode Performance Comparisons							
N	M	S	P	X	EE588	Thesis	Percent
states	measurements	noises	controls	terms	Times	Times	Decrease
64	32	32	16	1	502	242.5	51
64	32	32	16	2	528	266.6	50
40	20	20	0	1	125	62.4	50
40	20	20	0	2	131	68.8	47
20	6	6	0	1	15	7.6	49
20	6	6	0	2	16	8.5	47
16	1	6	0	1	6.9	3.6	48
16	1	6	0	2	7.4	4.1	46

Table 7.1. Microcode Performance Comparisons (Times in msecs)

total number of floating point operations required for a 64–state Kalman Filter with one term in the series expansion of $\Phi(t_i, t_{i-1})$ is 2.902×10^6 which, for a 25 MHz clock, yields an execution time of 116 msecs. The microcode developed as part of the research requires 242.5 msecs implying that only 52% of the time is spent performing ancillary operations.

Testing the Kalman Filter microcode was made possible with the development of the RTL simulator written in C. The fact that the RTL simulator is written in C allows the simulator to be ported to many different computers. The entire Kalman Filter microcode program was developed on a personal PC while the major simulation runs were performed on an Elxsi 6400. Using the Elxsi 6400, one nanosecond of simulation time requires 25.5 microseconds of real time meaning a complete simulation of a 64–state Kalman Filter requires no more than 100 minutes. Other benefits realized from developing the RTL simulator with C are the abilities to place software hooks in the simulator to capture data structures during the simulation. The Kalman Filter algorithm performs some very complex manipulations on its data structures, manipulations that couldn't readily be followed by reviewing the simulator generated state files.

Developing the FPASP register transfer level simulator with VHDL has demonstrated the versatility of this hardware design language. The VHDL design entities developed to model the FPASP components emulate the operating characteristics of the hardware much more accurately than subroutines written in a programming language such as C. Ensuring that the design entities map accurately to hardware they are modeling will ease the process of updating the VHDL simulator should the FPASP design be modified. Writing the code for the VHDL simulator was considerably more difficult than writing the code for the RTL simulator. The most difficult part of the code development was learning the VHDL constructs of signal assignment and signal resolution, and synchronizing the execution of the individual design entities. The VHDL simulator contains 14 design entities and six packages requiring 5300 lines of code, which includes all comments and module headers (Koc88) . This compares to the 2000 lines of code, again including all comments and module headers, to write the C language RTL simulator. Realizing that VHDL is highly structured language and C is not, the difference is easy to understand. The only disappointing aspect of the VHDL simulator is its slow execution times. Running the VHDL simulator on a VAX 8800, one nanosecond of simulation time requires 125 milliseconds of real time. Simulating a simple 4-state Kalman Filter requires approximately 18 hours to perform 5000 simulation cycles. This compares to five minutes using the RTL simulator on an Elxsi 6400.

The VHDL design entities developed to model the FPASP components are rather complex, and defied any attempts to provide accurate written descriptions. In response, the Entity Structure Diagram was developed as a tool to graphically depict the structure of a VHDL design entity. Included in the scope of this thesis, was the requirement to accurately model the FPASP. Even though the developer of the simulator may feel the model is accurate, he has the responsibility to convince others. The Entity Structure Diagram has made this task easier by providing the means to demonstrate the mapping between the software and the hardware that is easily understood.

7.2 Recommendations

A major portion of the time devoted to this thesis went into the development of the Kalman Filter microcode. Approximately two months were required to develop, debug, optimize, and test the code. Starting with only minimal exposure to the FPASP micro-instruction set, it could be argued that a majority of the time spent was on the front end of a learning curve. In retrospect, after becoming intimately familiar with the FPASP micro-instruction set, if the Kalman Filter microcode were to be rewritten, it would still require a considerable amount of time. The same Kalman Filter algorithm was implemented with C as part of the microcode development and required only two evenings. An automated tool, for translating a higher-order language description of an algorithm into a format compatible with the GMAT assembler, would drastically reduce the time required to develop FPASP microcode.

The software tools developed as part of this research effort, in particular the RTL simulator, provide an excellent basis for follow-on class projects. The RTL simulator is rather primitive, yet it should be easy to integrate into a full microcode application environment. Even without a formal development environment, the RTL simulator can provide EE588 students the opportunity to program and execute microcode routines. The microcode routines could then be collected into a library of routines for possible programming into the FPASP.

A considerable interest is being placed in software development tools that translate graphical data flow charts (flow charts, structure charts, etc.) to some form of higher-order language that can be compiled and executed. An interesting software engineering project would be a feasibility study into translating design entities developed using the Entity Structure Diagram directly to VHDL. The Entity Structure Diagram in its current form lacks the detail to express all the constructs of a design entity. Yet it does demonstrate that the capability is there.

One of the benefits of using VHDL to model a VLSI architecture is that VHDL allows a rich combination abstraction levels to exist concurrently in the same simulator. The VHDL simulator developed in this research effort models the FPASP at the register level using subroutines to emulate component functions. As the macro cell designs of the FPASP components are completed and verified, the design entity for each component should be updated and reinstantiated into the simulator.

F.P.ASP Architecture Specification
Microword Format
rev 3.0: 14 December 1988

Upper ROM

64 Bits

A BUS SEL (5)	B BUS SEL (5)	C BUS SEL (5)	ALU SEL (4)	SHIFT CTRL (4)	FPMULT CTRL (3)	FPADD CTRL (3)	INC CTRL (3)	AB PTR CTRL (3)	MBR CTRL (2)	MAR CTRL (2)	E BUS TIE (1)	E BUS CTRL (2)	MEM CS (1)	MEM WE (1)	MEM OE (1)	FUNCT ROM (3)

C BUS TIE (1)	LIT INS CTRL (4)	MUX SELECT (6)	BRANCH CTRL (2)	MACRO SEL (3)

Lower ROM

64 Bits

A BUS SEL (5)	B BUS SEL (5)	C BUS SEL (5)	ALU SEL (4)	SHIFT CTRL (4)	BARREL SET-UP (2)	BARREL SHIFT (5)	INC CTRL (3)	AB PTR CTRL (3)	MBR CTRL (2)	MAR CTRL (2)	E BUS SEL (2)	A BUS TIE (1)	B BUS TIE (1)	MEM CS (1)	MEM WE (1)

MEM OE (1)	DONE BIT (1)	NEXT AD LITERAL (16)

Revision 3.0, 14 December 1988

bits 0-4		Upper A Bus Select
00000:	NOP1	(no drive - DEFAULT)
00001:	AU=R1	(upper A bus = upper R1)
00010:	AU=R2	
⋮		
11001:	AU=R25	
11010:	AU=IN1	(upper A bus = upper INC1)
11011:	AU=IN2	
11100:	AU=IN3	
11101:	AU=APT	(upper A bus = A pointer)
11110:	AU=BPT	
11111:	AU=MBR	(upper A bus = upper MBR)
bits 5-9		Upper B Bus Select
00000:	NOP2	
00001:	BU=R1	
00010:	BU=R2	
⋮		
11001:	BU=R25	
11010:	BU=IN1	
11011:	BU=IN2	
11100:	BU=IN3	
11101:	BU=FP*	(upper B bus = result FP Multiplier, MSBs)
11110:	BU=FP+	(upper B bus = result FP Adder, MSBs)
11111:	BU=MBR	
bits 10-14		Upper C Bus Select
00000:	NOP3	
00001:	R1=CU	(upper R1 loads from upper C bus)
00010:	R2=CU	
⋮		
11001:	R25=CU	
11010:	IN1=CU	(upper INC1 loads from upper C bus)
11011:	IN2=CU	
11100:	IN3=CU	

11101:	APT=CU	(A pointer loads from upper C bus)
11110:	BPT=CU	
11111:	MBR=CU	(upper MBR loads from upper C bus)

bits 15-18 Upper ALU Select

0000:	NOP4	
0000:	MOVUN	(SHIFTER_INPUT=A, flags unaffected)
0001:	ORU	(SHIFTER_INPUT=A or B)
0010:	ANDU	(SHIFTER_INPUT=A and B)
0011:	XORU	(SHIFTER_INPUT=A xor B)
0100:	MOVU	(SHIFTER_INPUT=A, affects flags)
0101:	NANDU	(SHIFTER_INTPUT=A nand B)
0110:	NORU	(SHIFTER_INPUT=A nor B)
0111:	NOTU	(SHIFTER_INPUT=A')
1000:	INCU	(SHIFTER_INPUT=A + 1)
1001:	SETU	(Set Carry Flip Flop)
1010:	ADCU	(SHIFTER_INPUT=A + B + cy flip-flop)
1011:	ADDU	(SHIFTER_INPUT=A + B)
1100:	NEGAU	(SHIFTER_INTPUT= −A)
1101:	SUBU	(SHIFTER_INPUT =A − B)
1110:	SWBU	(SHIFTER_INPUT =A − B − cy flip-flop)
1111:	DECU	(SHIFTER_INPUT =A − 1)

bits 19-22 Upper Shifter Control (INPUTS DIRECTLY FROM ALU)

0000:	NOP5	(Shifter does not drive C bus)
0001:	GNDCU	(C = 0, shift flags unaffected)
0010:	PASSU	(C = SHIFTER_INPUT, shift flags unaffected)
0011:	SLOTU	(SHL with SH_OUT into LSB)
0100:	SLMSU	(circulate left with MSB into LSB)
0101:	SLCYU	(SHL with CY out of ALU into LSB)
0110:	SL0U	(SHL with 0 into LSB)
0111:	SL1U	(SHL with 1 into LSB)
1000:	SRLSU	(circulate right with LSB into MSB)
1001:	SRCFU	(SHR with carry flip-flop into MSB)
1010:	SRSU	(SHR with SIGN FF into MSB)
1011:	SROTU	(SHR with SH_OUT FF into MSB)
1100:	SRSEU	(SHR with sign extension)
1101:	SRCYU	(SHR with CY out of ALU into MSB)
1110:	SR0U	(SHR with 0 into MSB)
1111:	SR1U	(SHR with 1 into MSB)

bits 23-25 Floating Point Multiplier Control

000:	NOP6	
000:	FP*	(do floating point multiply - DEFAULT)
001:	FP*D	(drive results onto C bus, latch flags)
010:	FP*L	(load A, B bus inputs)
011:	FP*PD	(load A,B bus inputs, drive C bus, latch flags)
100:	INT*	(do integer multiply, must be selected during INT*)
101:	INT*D	(drive product onto C bus)
110:	INT*L	(load 32 bit integers from A & B busses)
111:	INT*LD	

bits 26-28 Floating Point Adder Control

000:	NOP7	
000:	FP+	(do floating point add - DEFAULT)
001:	FP+D	(drive results onto C bus, latch flags)
010:	FP+L	(load A, B bus inputs)
011:	FP+LD	(load A, B inputs, drive C bus, latch flags)
100:	FP-	(do floating point subtaction)
101:	FP-D	(drive C bus while doing subtract)
110:	FP-L	(load new inputs while doing subtract)
111:	FP-LD	(load and drive while doing subtract)

bits 29-31 Upper Incrementable Registers Control

000:	NOP8	(no increment - DEFAULT)
001:	UIN1+	(increment upper INC1)
010:	UIN2+	(increment upper INC2)
100:	UIN3+	(increment upper INC3)

bits 32-34 A, B Pointer Controls

000:	NOP9	(no change - DEFAULT)
001:	AIN_L	(load A increment size)
010:	BIN_L	(load B increment size)
011:	ABIN_L	(load A, B increment sizes)
100:	APT+	(increment A pointer)
101:	A+_AL	(increment A pointer, load A increment)
110:	B+_BL	(increment B pointer, load B increment)
111:	BPT+	(increment B pointer)

bits 35-36 Upper Memory Buffer Registers Control

00:	NOP10	(no action - DEFAULT)
01:	R1=DU	(load upper R1 from upper D bus)
10:	R2=DU	(load upper R2 from upper D bus)

11:	MB=DU	(load upper MBR from upper D bus)

bits 37-38 Upper Memory Address Register Control

00:	NOP37	(no action - DEFAULT)
01:	MAR=CU	(drive upper addr bus and load from C bus)
10:	MAR=EU	(drive upper addr bus and load from E bus)
11:	MARU+	(increment upper MAR)

bit 39 E Bus Tie

0:	(no drive - DEFAULT)
1:	(tie upper and lower E busses together)

bits 40-41 Upper E Bus Select

00:	NOP12	(no drive -DEFAULT)
01:	E=APT	(drive A pointer onto upper E bus)
10:	E=BPT	(drive B pointer onto upper E bus)
11:	UADRZ	(Upper address and memory control pads to hiZ)

bit 42 Upper Memory Chip Select Bar

0:	NOP13	(active low)
1:	UCS_H	

bit 43 Upper Memory Write Enable Bar

0:	WEBU	(active low)
1:	NOP14	(data pads to hiZ, input only - DEFAULT)

bit 44 Upper Memory Chip Output Enable Bar

0:	NOP15	(active low, output enabled - DEFAULT)
1:	UOE_H	

bits 45-47 Function ROM Select

000:	NOP16	(ROM does not drive C bus - DEFAULT)
001:	SQR	(Square Root)
010:	RCP	(Reciprocal)
011:	user defined	(LPROM)
100:	user defined	(LPROM)
101:	user defined	(LPROM)
110:	user defined	(LPROM)

111:	user defined (LPROM)	

bit 48 C Bus Tie

0:	NOP17
CTIE	(tie both upper and lower C busses together)

bits 49-52 Literal Inserter Control

0000:	NOP18	(Shifter does not drive C bus - DEFAULT)
0001:	SAV1	(flags set1 to upper MSBs, LSBs no change)
0010:	SAV2	(flags set2 to upper MSBs, LSBs zeroed)
0011:	SAV3	(flags set3 to upper LSBs, MSBs no change)
0100:	ILZL	
0101:	IMNL	1st character: Insert literal
0110:	IMZL	
0111:	ILNL	2nd character: LSBs or MSBs get literal
1000:	ILZU	
1001:	IMNU	3rd character: Zero other half or no change
1010:	IMZU	
1011:	ILNU	4th character: UPPER/LOWER/BOTH busses get literal
1100:	ILZB	
1101:	IMNB	The mnemonic ILZL means Insert the
1110:	IMZB	literal on the LSBs and Zero out the MSBs
1111:	ILNB	of the lower A bus only.

bits 53-58 Conditional Multiplexer Select

000000: FAL	(unconditionally false - DEFAULT)
000001: IO1	(I/O interrupt level 1)
000010: IO2	(I/O interrupt level 2)
000011: MZ	(multiplier zero)
000100: MOVF	(mult overflow/int result more than 32 bits)
000101: MUN	(multiplier inderflow)
000110: MNAN	(multiplier NaN - Not a Number)
000111: MDEN	(multiplier denormaliztion trap)
001000: AZ	(adder zero)
001001: AOVF	(adder overflow)
001010: ANAN	(adder NaN - Not a Number)
001011: ADIF	(inputs to adder differ by more than 2^{64})
001100: TRPS	(MOVF + MNAN + AOVF + ANAN + UALU0 = 1)
001101: UIN1Z	(upper INC1 = 0)
001110: UIN2Z	(upper INC2 = 0)
001111: UIN3Z	(upper INC3 = 0)
010000: USO	(upper shifters shifted out bit = 0)

010001: LIN1Z	(lower INC1 = 0)
010010: LIN2Z	(lower INC2 = 0)
010011: LIN3Z	(lower INC3 = 0)
010100: UALUZ	(upper ALU zero)
010101: UALUN	(upper ALU negative)
010110: UALUO	(upper ALU overflow)
010111: UALUC	(upper ALU carry)
011000: unused	
011001: unused	
011010: unused	
011011: UEVN	(integer on upper C bus is even - LSB = 0)
011100: UR1_0	(UR1 bit 0 = 1)
011101: UR1_1	(UR1 bit 1 = 1)
011110: UR1_2	(UR1 bit 2 = 1)
011111: UR1_3	(UR1 bit 3 = 1)
100000: TRU	(unconditionally true)
100001: NIO1	(not I/O interrupt level 1)
100010: NIO2	(not I/O interrupt level 2)
100011: NMZ	(not multiplier zero)
100100: NMOVF	(not multiplier overflow)
100101: NMUN	(not multiplier underflow)
100110: NMNAN	(not multiplier NaN)
100111: NMDEN	(not multiplier denormalization trap)
101000: NAZ	(not adder zero)
101001: NAOVF	(not adder overflow)
101010: NANAN	(not adder NaN)
101011: NADIF	(inputs to adder do not differ by more than 2^{64})
101100: NTRPS	(MOVF + MNAN + AOVF + ANAN + UALU0 = 0)
101101: UIN1N	(upper INC1 not = 0)
101110: UIN2N	(upper INC2 not = 0)
101111: UIN3N	(upper INC3 not = 0)
110000: LSO	(lower shifter's shifted out bit = 0)
110001: LIN1N	(lower IN1 not = 0)
110010: LIN2N	(lower IN2 not = 0)
110011: LIN2N	(lower IN3 not = 0)
110100: LALUZ	(lower ALU zero)
110101: LALUN	(lower ALU negative)
110110: LALUO	(lower ALU overflow)
110111: LALUC	(lower ALU carry)
111000: unused	
111001: unused	
111010: unused	
111011: LEVN	(integer on lower C bus is even - LSB = 0)
111100: UR1_28	(UR1 bit 28 = 1)
111101: UR1_29	(UR1 bit 29 = 1)

111110:	UR1_30	(UR1 bit 30 = 1)
111111:	UR1_31	(UR1 bit 31 = 1)

bits 59-60 Branch Control

00:	BR	(conditional branch - DEFAULT)
01:	RET	(conditional return)
10:	CALL	(conditional call)
11:	MAP	(uncondotionally use the MAP next address)

bits 61-63 Macrocode Support Mux Selects

000:	NOP19	(DEFAULT)
001:	RSEL	(override register select/bus tie fields)
010:	BRSEL	(override condition mux select field)
011:	SRSEL	(override Ebus/Etie ctrls, reg sel, and bus tie fields)
100:	ALSEL	(override ALU/Shift, reg sel, and bus ties)
101:	SHSEL	(override Barrel shift ctrl, reg sel, and bus ties)
110:	ISEL	(override increment and pointer ctrl fields)
111:	unused	

A.2 Lower ROM Fields

bits 0-4 Lower A Bus Select

00000:	NOP1	(no drive - DEFAULT)
00001:	AL=R1	(lower A bus = lower R1)
00010:	AL=R2	
⋮		
11001:	AL=R25	
11010:	AL=IN1	(lower A bus = lower INC1)
11011:	AL=IN2	
11100:	AL=IN3	
11101:	AL=CPT	(lower A bus = C pointer)
11110:	AL=DPT	
11111:	AL=MBR	(lower A bus = lower MBR)

bits 5-9 Upper B Bus Select

00000:	NOP2	
00001:	BL=R1	
00010:	BL=R2	
⋮		
11001:	BL=R25	
11010:	BL=IN1	
11011:	BL=IN2	
11100:	BL=IN3	
11101:	BL=FP*	(lower B bus = result FP Multiplier, MSBs)
11110:	BL=FP+	(lower B bus = result FP Adder, MSBs)
11111:	BL=MBR	

bits 10-14 Lower C Bus Select

00000:	NOP3	
00001:	R1=CL	(lower R1 loads from lower C bus)
00010:	R2=CL	
⋮		
11001:	R25=CL	
11010:	IN1=CL	(lower INC1 loads from lower C bus)
11011:	IN2=CL	
11100:	IN3=CL	
11101:	CPT=CL	(C pointer loads from lower C bus)
11110:	DPT=CL	
11111:	MBR=CL	(lower MBR loads from lower C bus)

bits 15-18		Lower ALU Select
0000:	NOP4	
0000:	MOVLN	(SHIFTER_INPUT=A, flags unaffected)
0001:	ORL	(SHIFTER_INPUT=A or B)
0010:	ANDL	(SHIFTER_INPUT=A and B)
0011:	XORL	(SHIFTER_INPUT=A xor B)
0100:	MOVL	(SHIFTER_INPUT=A, affects flags)
0101:	NANDL	(SHIFTER_INTPUT=A nand B)
0110:	NORL	(SHIFTER_INPUT=A nor B)
0111:	NOTL	(SHIFTER_INPUT=A')
1000:	INCL	(SHIFTER_INPUT=A + 1)
1001:	SETL	(Set Carry Flip Flop)
1010:	ADCL	(SHIFTER_INPUT=A + B + cy flip-flop)
1011:	ADDL	(SHIFTER_INPUT=A + B)
1100:	NEGAL	(SHIFTER_INTPUT= −A)
1101:	SUBL	(SHIFTER_INPUT =A − B)
1110:	SWBL	(SHIFTER_INPUT =A − B − cy flip-flop)
1111:	DECL	(SHIFTER_INPUT =A − 1)

bits 19-22		Lower Shifter Control (INPUTS DIRECTLY FROM ALU)
0000:	NOP5	(Shifter does not drive C bus)
0001:	GNDCL	(C = 0, shift flags unaffected)
0010:	PASSL	(C = SHIFTER_INPUT, shift flags unaffected)
0011:	SLOTL	(SHL with SH_OUT into LSB)
0100:	SLMSL	(circulate left with MSB into LSB)
0101:	SLCYL	(SHL with CY out of ALU into LSB)
0110:	SL0L	(SHL with 0 into LSB)
0111:	SL1L	(SHL with 1 into LSB)
1000:	SRLSL	(circulate right with LSB into MSB)
1001:	SRCFL	(SHR with carry flip-flop into MSB)
1010:	SRSL	(SHR with SIGN FF into MSB)
1011:	SROTL	(SHR with SH_OUT FF into MSB)
1100:	SRSEL	(SHR with sign extension)
1101:	SRCYL	(SHR with CY out of ALU into MSB)
1110:	SR0L	(SHR with 0 into MSB)
1111:	SR1L	(SHR with 1 into MSB)

bits 23-24		Barrel Shifter Set-up
00:	SHROM	(shift using ROM control input - DEFAULT)
01:	SHREG	(shift using control register input)
10:	L_ROM	(load Bar ctrl reg, use ROM input to shift)

11:	L_REG	(load Bar ctrl reg, use old reg value to shift)

NOTE: If you just want to load the reg, use L_ROM and NOP25 below.

bits 25-29 Barrel Shifter Control

00000:	NOP25	(do not drive C bus - DEFAULT)
00001:	LCS1	(1 bit left circular shift)
00010:	LCS2	(2 bit left circular shift)
⋮		
11111:	LCS31	(31 bit left circular shift)

bits 30-32 Lower Incrementable Registers Control

000:	NOP26	(no increment - DEFAULT)
001:	LIN1+	(increment lower INC1)
010:	LIN2+	(increment lower INC2)
100:	LIN3+	(increment lower INC3)

bits 33-35 C, D Pointer Controls

000:	NOP27	(no change - DEFAULT)
001:	CIN_L	(load C increment size)
010:	DIN_L	(load D increment size)
011:	CDIN_L	(load C, D increment sizes)
100:	CPT+	(increment C pointer)
101:	C+_CL	(increment C pointer, load C increment)
110:	D+_DL	(increment D pointer, load D increment)
111:	DPT+	(increment D pointer)

bits 36-37 Lower Memory Buffer Registers Control

00:	NOP28	(no action - DEFAULT)
01:	R1=DL	(load lower R1 from lower D bus)
10:	R2=DL	(load lower R2 from lower D bus)
11:	MB=DL	(load lower MBR from lower D bus)

bits 38-39 Lower Memory Address Register Control

00:	NOP38	(no action - DEFAULT)
01:	MAR=CL	(drive lower addr bus and load from C bus)
10:	MAR=EL	(drive lower addr bus and load from E bus)
11:	MARL+	(increment lower MAR)

bits 40-41 Lower E Bus Select

00:	NOP29	(no drive -DEFAULT)
01:	E=CPT	(drive C pointer onto upper E bus)
10:	E=DPT	(drive D pointer onto upper E bus)
11:	LADRZ	(lower address and memory control pads to hiZ)
bit 42		A Bus Tie
0:	NOP30	
1:	ATIE	(tie upper and lower A busses together)
bit 43		B Bus Tie
0:	NOP31	
1:	BTIE	(tie upper and lower B busses together)
bit 44		Lower Memory Chip Select Bar
0:	NOP33	(active low)
1:	LCS_H	
bit 45		Lower Memory Write Enable Bar
0:	WEBL	(active low)
1:	NOP34	(data pads to hiZ, input only - DEFAULT)
bit 46		Lower Memory Chip Output Enable Bar
0:	NOP35	(active low, output enabled - DEFAULT)
1:	LOE_H	
bit 47		Done Flag
0:	NOP36	(no action - DEFAULT)
1:	DONE	(raise done flag)
bits 48-63		Next Address/Literal Field

(Literals are composed of the bit pattern preceeded by #)

Appendix B. *Entity Structure Diagram Syntax*

The Enitity Structure Diagram was developed to provide some means to graphically depict a VHDL design entity. A rather generic Entity Structure Diagram is shown below in Figure B.1.

Associated with every design entity is a port list that names the signals imported to or exported from the entity. Associated with every signal listed within the port list is a mode. That mode can be either in, out, or inout. The Entity Structure Diagram shows this port, as well as the signals in the port list, and each signal's associated mode. In Figure B.1, the port of the design entity is clearly identified, as are the signals associated with the port. The signal modes are also clearly identified. Signals X and Z are of mode in, signal Y is of mode out, and signals AA and BB are of mode inout. What the diagram doesn't signify is the Type of the signals. The Entity Structure Diagram is not concerned with the fine details of a design entity, but rather attempts to provide an abstraction of the overall entity. Details, such as the Types of individual signals, can be referenced from the source document.

The signals listed in the port are connected internally to a logical network bus. This facilitates illustrating the logical data flow within the architecture. The logical busses are shown in Figure B.1, and run vertically up and down the left side of the diagram. Emanating to and from the logical busses are signal lines. Each signal line has an arrow head to indicate the direction of data flow.

The basic constructs of the design entity are the block statements, and the process statements. Associated with every block statement is a guard expression, and with every process statement is a sensitivity list. The Entity Structure Diagram depicts blocks and processes as rectangles with header bars. Each header bar is labeled as block, or process, and the header bar itself represents the guard expression, or the sensitivity list respectively. Alternatively, the header bar could contain the label name of the block or process that it represents. In complicated architectures, this would make it easier to identify the structure that it is representing. A signal line terminating in the header of a block implies the signal is part of the guard expression for that block (refer to the source document for the guard expression). Likewise, a signal line terminating in the header of a process implies the process is sensitive to that signal.

Block number one in Figure B.1 has two signals, X and AA, terminating in its header. The implication here is that X and AA are used in the guard expression of the block, and the concurrent

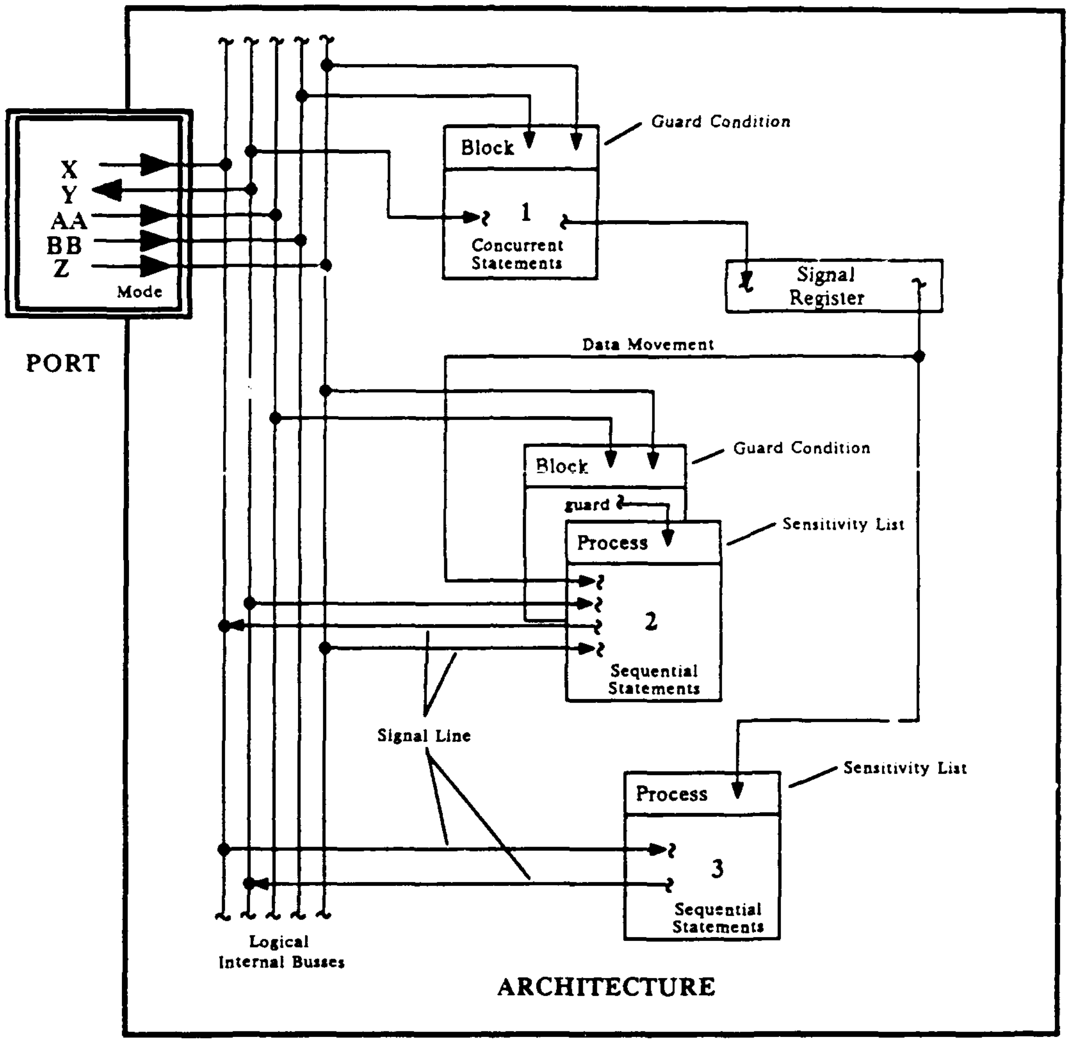


Figure B.1. Entity Structure Diagram Syntax

statements within the body of the block are enabled according to specific signal values on X and AA. Process number three on the other hand, has the output of a locally declared signal register in its sensitivity list. In this case, the sequential statements within the body of the process will execute whenever a new value is loaded into the signal register. The exception here is the case where the TYPE of signal register is some type of an array, and only a slice of the array is actually listed in the sensitivity list of process number three. Referencing the source document will clear this question up. The final example in Figure B.1 is the block, process pair number two. This is a special case (there could be others) where the sensitivity list of the nested process lists the single implicit signal GUARD. The signals AA and Z terminate in the header of the block statement, and, depending on the values of AA and Z, when the condition is True the impicit signal GUARD is changed. The nested process senses this executes the sequential statements within its body.

The remainder of the diagram contains signal lines emanating and terminating from the bodies of the blocks and processes. These signal lines represent the logical flow of data that has been processed by blocks and processes. Process number three for example, is controlled by changes in the signal register causing some action to be performed on signal X and outputed on signal Y.

The Entity Structure Diagram was designed primarily to provide a higher level abstractional view of a design entity that could be graphically depicted. The actual instructions contained within guard expressions, and the architectures of the blocks and processes are left to the source document, and would only clutter the diagram.

Bibliography

Arm89. Armstrong, James R., *Chip Level Modeling with VHDL*, Prentice Hall, Englewood Cliffs, New Jersey, 1989

Bor88. Boriello, Gaetano and Detjens, Ewald, "High-Level Synthesis: Current Status and Future Directions," *Proceedings of the IEEE 1988 Design Automation Conference*, IEEE Press, March 1988.

Cav84. Cavanaugh, Joseph J. F., *Digital Computer Arithmetic*, McGraw-Hill, New York, New York, 1984.

Coe89. Coelho, David R., *The VHDL Handbook*, Vantage Analysis Systems, 1989.

Com88. Comtois, John Henry, "Architecture and Design for a Laser Programmable Double Precision Floating Point Application Specific Processor," *MS Thesis*,AFIT/GE/ENG/88-5, School of Engineering, Air Force Institute of Technology (AU), December 1988.

Fle88. Fleckenstein, Donald C., chairman, "IEEE Standard VHDL Language Reference Manual," Institute of Electrical and Electronics Engineers, INC., New York, New York, 1988.

Fuj85. Fujiwara, H., *Logic Testing and Design for Testability*, MIT Press, Cambridge, Massachussettes, 1985.

Gal87. Gallagher, David M., "Rapid Prototyping of Application Specific Processors," *MS Thesis*, AFIT/GE/ENG/87D-19, School of Engineering, Air Force Institute of Technology (AU), December 1987.

Int85. Inter. etrics, INc. *VHDL User's Manual: Volume I Tutorial.* US Air FOrces Contract F33615-83-C-1003. Bethesda Md., 1 August 1985.

Hau87. Hauser, Robert S., "Design and Implementation of a VLSI Prime Factor Algorithm Processor," *MS Thesis*, AFIT/GCE/ENG/87D- 5, School of Engineering, Air Force Institute of Technology (AU), December 1987.

Hus88. Huson, March, Harding, and Asilva, "Kalman Filter Update Project," EE588 Class Project, School of Engineering, Air Force Institute of Tech ·ology (AU), August 1988.

Koc88. Koch, William E., "FPASP Simulator Source Code, Kalman Filter Microcode and Validation Results," School of Engineering, Air Force Institute of Technology (AU), October 1989.

Lew89. Lewantowicz, Lt Col Zdzislaw H., Deputy Department Head, Department of Electrical and Computer Engineering, Personal Interviews, Air Force Institute of Technology (AU), 16 June to 13 October 1989.

Lin88. Linderman, Richard W., editor, "Compendium of Project Reports Submitted for EENG 588: Computer Systems Architecture Summer Quarter 1988," *AFIT Technical Report*, Air Force Institue of Technology (AU), October 1988.

Lin88. Linderman, Richard W. and Lewantowicz, Lt Col Zdzislaw H., "Kalman Filter Propagation and Update," *Student handout*, Air Force Institute of Technology Department of Electrical and Computer Engineering, Wright-Patterson AFB, Ohio, 1988.

May79. Maybeck, Peter S., *Stochastic Models, Estimation, and Control*, Volume I, Academic Press, San Diego, California, 1979.

May82. Maybeck, Peter S., *Stochastic Models, Estimation, and Control*, Volume II, Academic Press, San Diego, California, 1982.

Sno89. Snodgrass, F. Britt, Graduate Student, Stochastic Estimation and Control Theory, Personal Interviews, School of Engineering, Air Force Institute of Technology (AU), October, 1989.

Scr89. Scriber, Micheal, " ," *MS Thesis*, AFIT/CE/ENG/89D, School of Engineering, Air Force Institute of Technology (AU), December 1989.

Sum81. Sumney, Larry W., "Government Interest and Invovlement in Design Automation Defvelopment - The VHSIC Perspective," *IEEE Design Automation Conference*, pp. 344 - 346, October 1981.

Til88. Tillie, John L., "Laser Programmable Read Only Memories," *MS Thesis*, AFIT/GE/ENG/88D-56, School of Engineering, Air Force Institute of TEchnology (AU), December 1988.

Wai87. Waite, Mitchell, Stephen Prata, and Donald Martin., *C Primer Plus*, Howard W. Sams and Company, Indianapolis, Indiana, 1987.

Vita

Capt William E. Koch [redacted] Shortly after graduation, Capt Koch enlisted in the US Air Force. After serving several tours in the Far East, he was accepted into the Airman's Education and Commissioning Program from which he received a B.S.E.E from Texas A&M University. Capt Koch received his commission in the US Air Force through the Officer Training School. His first assignment after commissioning was to the Site Activation Task Force (SATAF) for the Ballistic Missile Office at Grand Forks, North Dakota. Following that assignment, he was assigned to the SATAF at Cheyenne Wyoming where he was a field engineer for the installation of the Peacekeeper missile system. He remained in Cheyenne until his acceptance into the AFIT. Following his graduation from AFIT, Capt Koch will be assigned to the 513th Test Squadron/SAC at Omaha Nebraska.

[redacted]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE				
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/ENG/89D-3		5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433-6583		7b. ADDRESS (City, State, and ZIP Code)		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 6585TG/GD (CIGTF)	8b. OFFICE SYMBOL (If applicable) GDN	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code) Holloman AFB, NM 88330		10. SOURCE OF FUNDING NUMBERS		
		PROGRAM ELEMENT NO.	PROJECT NO	TASK NO / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Development of the Kalman Filter Application and a VHDL Model for the AFIT Floating Point Application Specific Processor (FPASP) (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
William E. Koch, B.S., Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Thesis	FROM ____ TO ____	1989 December	156

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Computer Architecture, VLSI Microprocessor, VHSIC Hardware Design Language (VHDL)
09	01		

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Chairman: Keith R. Jones, Captain, USAF
Instructor

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Keith R. Jones, Captain, USAF	513-255-3576	AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

19. Abstract

The Air Force Institute of Technology (AFIT) is conducting research that will lead to the development of a Floating Point Application Specific Processor (FPASP). The FPASP architecture is designed around two independent 32 bit data paths that work in tandem to support full IEEE double precision floating point operations, or that can work independently for 32 bit integer processing. Designed to operate at 25 MHz, the FPASP will be capable of performing 25 million floating point operations per second.

A rapid prototyping methodology has been developed for the FPASP. In this methodology, a user identifies an application that could benefit from a VLSI solution. An algorithm of the application is translated into FPASP microcode which can then be programmed into the Laser Programmable Read Only Memory (LPROM) of a blank FPASP. The programmed FPASP can then be mounted on a circuit card and installed in a host system where it would function as a hardware accelerator supporting the user application.

This thesis supports AFIT's continuing development of the FPASP in two areas. In the first part of this thesis, a user application, the Kalman Filter algorithm, is translated into FPASP microcode for programming into the FPASP. To support the microcode generation, verification, and validation of the Kalman Filter microcode program, a software model of the register transfer level design of the FPASP was developed using the C programming language. The successful development of the Kalman Filter microcode program demonstrated not only the FPASP's abilities to implement a complex algorithm, but also the initial phases of the rapid prototyping methodology. In the second part of this thesis, the feasibility of using the VHSIC (Very High Speed Integrated Circuitry) Hardware Design Language (VHDL) to model a complete system is demonstrated by developing a register transfer level model of the FPASP. Using the VHDL model of the FPASP, it was possible to validate the design concepts used in the development of the FPASP, as well as providing support for architectural enhancements by allowing enhancements to be simulated prior to fabrication changes.